SCHOOL OF INDUSTRIAL & ICT ENGINEERING

MEDICAL AID SYSTEM FOR ALZHEIMER'S DIAGNOSIS USING MACHINE LEARNING

BACHELOR'S DEGREE IN COMPUTER SCIENCE

FINAL DEGREE PROJECT

Jon Urtasun Urra

Daniel Paternain Dallo and Mikel Galar Idoate

Pamplona, 29 January 2021

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Abstract

Artificial Intelligence (AI), and precisely Machine Learning (ML), has been proved to be a considerably useful method in a wide variety of fields. One of the fields that has improved more is the computer vision. Computers can now apply ML techniques in order to solve tasks, such as image classification, which were inconceivable in the past. This led into a huge impact on the medicine field, which has increasingly ameliorated over the recent years. The purpose of this project is to prove whether ML techniques are able to successfully diagnose Alzheimer's disease or not.

Keywords

Alzheimer, machine learning, MRI, dataset, convolutional neural network, SVM.

Index

Figure index

Table index

1. Introduction

Alzheimer's disease is an irreversible, progressive brain disorder that slowly destroys memory and thinking skills and it is the most common cause of dementia among older adults [1].

Despite the undeniable fact that major improvements have been made in medicine in the recent years, Alzheimer's diagnosis is a tedious process, which involves several phases and people. It goes through examining the patient's medical history, carrying out medical tests and performing brain scans. This process is begun by the patient's primary doctor, but it is referred to specialists who can provide a detailed diagnosis [2].

However, Artificial Intelligence (AI) has gone through a much deeper improvement – computers can now solve problems that were inconceivable. One of the techniques inside AI is Machine Learning (ML), which is the study of computer algorithms that improve automatically through experience.

This led to the emergence of Deep Learning (DL), a subfield of ML that deals with algorithms inspired by the structure and function of the brain called Artificial Neural Networks (ANNs) [3]. DL has proved to have a better performance than ML in a lot of problems, and it is capable of scaling the performance with big amount of data. Nevertheless, it requires large data, takes longer to train and depends on the GPU (Graphics Processing Unit) to train properly.

These ANNs consist in a collection of connected units or nodes called "artificial neurons", which are capable of receiving input data and generating an output [4]. This collection is divided in different layers, the input layer, the hidden layers and the output layer. Artificial neurons can be stacked in order to form increasingly deeper networks, augmenting the number of hidden layers.

In addition, there are different types of ANNs. Shallow Neural Networks (SNNs) have a small number of hidden layers, whereas Deep Neural Networks (DNNs) are built with a higher number of hidden layers. Then, new architectures such as Convolutional Neural Networks (CNNs) came up.

From all of these, we are going to focus on CNNs, which receive an image as the input, employ a mathematical operation called convolution in at least one of the hidden layers, and output the class of the image.

The purpose of this project is to prove whether ML techniques are able to successfully diagnose Alzheimer's disease or not. The idea is making a medical aid system that can detect Alzheimer's disease in early stages to prevent it from reaching further ones. In order to accomplish this objective, we are going to divide this project in four main chapters:

- In the first chapter, we are going to overview the tools and technologies that are going to be used in the development of this project.
- In the second chapter, we are going to present the project's life cycle.
- In the third chapter, we are going to explain all the project's development stages. Firstly, we will make a research in order to have a basic knowledge of the disease. Then we will go through the creation of all the datasets. After that, we will create the ML models, test them and optimize them so as to obtain the best possible results. Finally, we will expound the results.
- In the last chapter, we will analyse and draw conclusions about the results to see if they are able to corroborate the thesis.

From now on, when we mention machine learning, it is referred to traditional machine learning techniques, the non-deep learning ones.

2. Tools and technologies

Taking into account the background and objectives of this project, we proceed to select which tools and technologies are the most appropriate in order to carry out its development.

2.1. Programming language

A programming language is used to create software programs via a certain syntax. There are several programming languages for the resolution of artificial intelligence problems such as R, Julia or Python. Python is our selected programming language because it has a wide variety of libraries that make the process of developing a project much more efficient.

2.2. Development environment

2.2.1. Anaconda

Anaconda is a distribution of (in this case) the Python language for scientific computing (data science, machine learning applications, etc) that aims to simplify package management and deployment, and it is a platform that makes machine learning faster and easier [5].

One of the properties that make Anaconda useful to solve AI problems is the ease to create environments. Opening the Anaconda Navigator, we find a tab on the left side named Environments (where we can create and import our environments), and we can also install only the libraries we are going to use for the project.

2.2.2. Jupyter Notebook

Once we have installed Anaconda, we are able to initiate Jupyter Notebook [6], which is an open-source web application that allows the creation of live code.

Jupyter permits the editing and execution of notebook documents through any web navigator and it can be executed in a local directory without Internet connection. There are other development environments, but we are going to use this one for simplicity.

2.3. Python modules and libraries

After creating the Anaconda environment, we need to see which libraries might be useful to achieve the project goals. This environment does not have the libraries we want, but when we need them, we just have to search and install them and add the corresponding imports.

2.3.1. OS

OS is a Python module that provides functions for interacting with the operating system, used for moving between directories in order to access data [7].

2.3.2. OpenCV

OpenCV is a Python library designed to solve computer vision problems. It is a highly optimized library with focus on real-time applications [8]. Since we are working with image datasets, OpenCV allows us to read them so as to create our training and test sets.

2.3.3. NumPy

NumPy is a Python library which adds support for multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays [9].

2.3.4. Math

Math is a Python module which provides access to the mathematical functions defined by the C standard [10].

2.3.5. CSV

CSV is a Python module which implements classes to read and write tabular data in CSV format [11]. We can use this library to safe our trained models or data in an external file as we might want to access it later.

2.3.6. Matplotlib

Matplotlib is a Python library for creating static, animated and interactive visualizations [12], which can be useful to expose data in a clear and understandable way.

2.3.7. Itertools

Itertools is a Python module that provides functions of complex iterators, working as a fast, memory-efficient tool [13].

2.3.8. Scikit-learn

Scikit-learn is a Python library for machine learning which has simple and efficient tools for predictive data analysis [14]. This is the library we are going to use in order to create machine learning models.

2.3.9. Scikit-image

Scikit-image is a collection of algorithms for image processing [15]. In this case, we are going to use it later on for feature extraction.

2.3.10. TensorFlow

TensorFlow is an open-source platform for machine learning. It has a wide variety of tools, libraries and resources that allows the users to develop machine learning applications [16]. It is used to develop deep learning models.

2.3.11. Keras

Keras is a Python API for deep learning. It is built on top of TensorFlow 2.0, and it can be used to solve different problems [17]. In this project, we are going to use Keras so as to create deep learning models, specifically convolutional neural networks.

3. Life cycle

Before the beginning of the project development, we should decide which stages we are going to follow in order to keep everything organized and clear [18].

3.1. Data collection

The first stage is as simple as collecting data that might be useful so as to achieve the objectives of the project.

3.2. Dataset preparation and pre-processing

After obtaining the data we are going to work with, we need to adapt them in order to fit machine learning (ML). Structured and clean data allow us to get precise results from an applied ML model. We might need to go through substages like data labelling, data selection, data formatting, etc.

3.3. Dataset splitting

A dataset used for ML should be partitioned into at least two subsets – training and test sets. The training set is used to train a model and define its optimal parameters. The test set is used to evaluate the capability of the model after having been trained over the training set for generalization, in other words, the model's ability to identify patterns in unseen data.

There is a third optional set, the validation set, which purpose is to tune the model's hyperparameters, which are structural settings that cannot be directly learned from data.

The percentage of the selected data for the training, test and validation sets might vary depending on the problem, but it is usually established in around 60 % for the training set, 20 % for the test set and 20 % for the validation set. If the validation set is not used, the training set goes from 60 % to 90 % and the test set from 40 % to 10 % depending on the model and dataset size.

3.4. Modelling

As we are working in a classification problem, we are going to use supervised learning, which consists in training the models with a previously known data labelling in order to predict the class of a new image. This will be covered later on.

In this stage we are going to train numerous models (classifiers) for them to predict the labelling of unseen data to examine which of them provides the most accurate results.

We already have the training set, so we can start with the model training. This process “feeds” the algorithm with training data and it will output a model that is able to classify new data.

3.5. Model deployment

Once we get a good enough model for our problem, the next step is to put it into production use. This means that our model will have to face real cases, and our task is to monitor its performance.

3.6. Conclusion

In the last stage, as previously said, we will have to monitor if the model’s results correspond to performance requirements and improve it if needed. One of the most useful methods for doing this is displaying graphs with actualized performance results.

4. Development

Now that we know the stages we are going to follow, the tools, the technologies and the purpose of the project, we can begin with its development.

4.1. Previous research

Before building the dataset we are going to work with, we need to know more about Alzheimer's disease.

As previously said, Alzheimer's disease is an irreversible, progressive brain disorder that slowly destroys memory and thinking skills. We know that its diagnosis is a tedious process that involves several steps and people.

However, as we want to use machine learning techniques so as to successfully predict this disease, we might want to focus our attention on the steps where the doctor or specialist handles images. These steps are usually the last ones, after conducting medical and memory tests, and they are usually brain scans, such as computed tomography (CT), magnetic resonance imaging (MRI), or positron emission tomography (PET).

Furthermore, every person with Alzheimer's experiences the disease differently, but they tend to experience a similar trajectory from the beginning of the illness until its end. There are experts who use a simple model divided in three phases (early, moderate and end), while others make a deeper breakdown to have a better understanding of the progression of the illness.

Dr Barry Reisberg breaks the progression of Alzheimer's disease into seven stages [19]. This system has been adopted and used by healthcare providers as well as the Alzheimer's Association. This is a summary of the seven stages of Dr Reisberg's system:

4.1.1. Stage 1: No impairment

At this stage, Alzheimer's is not detectable and there are no symptoms of dementia.

4.1.2. Stage 2: Very mild decline

People who are at this stage of Alzheimer's might notice minor memory problems. However, they will still do well on memory tests and dementia might not be noticeable.

4.1.3. Stage 3: Mild decline

When someone reaches this stage, his or her family members and friends may begin to notice cognitive problems and, in addition, their performance on memory tests is worse. They also start to have problems in areas including organization and planning, remembering names of new people or finding the right word during a conversation.

4.1.4. Stage 4: Moderate decline

In stage four, symptoms of dementia are evident. People who are in this stage have problems with simple arithmetic, short-term memory, and they are unable to manage finance, pay bills and might forget about their life stories details.

4.1.5. Stage 5: Moderately severe decline

During the fifth stage, people start to need help in everyday activities. They have difficulties when dressing and they are unable to remember their personal details, like their own mobile phone number. On the contrary, they still can manage to carry on their normal routines, and they usually still recognise their family members.

4.1.6. Stage 6: Severe decline

At this stage, people need constant supervision and professional care. They are unaware of their surroundings. They are unable to recognize faces except for their closest friends and relatives, and unable to remember most details of their personal history. It also affects their behaviour, which is continuously changing, and they need assistance in daily activities, such as bathing.

4.1.7. Stage 7: Very severe decline

Stage seven is the final stage of Alzheimer's because the disease is a terminal illness. People in this stage lose the ability to communicate or respond to the environment. They need assistance in almost all the daily activities, in some cases they even lose the ability to swallow.

Now we have to see the availability of data and its labelling so as to see whether it corresponds to this stage breakdown or not. We are essentially interested in the early stages so that this medical aid system can help doctors in the premature diagnosis of Alzheimer's disease, which is fundamental in order to prescribe a treatment.

4.2. Dataset

Once we have enough knowledge of Alzheimer's disease, we can begin to look for data with the objective of building our datasets. Neurodegenerative diseases are frequently associated with structural changes in the brain. Magnetic Resonance Imaging (MRI) scans can show these variations and therefore can be useful for the Alzheimer's diagnosis [20]. As the purpose of the project is to see whether machine learning techniques can successfully diagnose this disease, the data we are looking for is a collection of MRI images that could be useful to satisfy this goal.

Taking into account the different stages of Alzheimer's seen in the previous section, we will look for datasets that have a similar division, focusing our attention on the first stages of the disease.

We will have to consider as well that the better the data is, the less we need to pre-process it. This means that if we can find an organized and clean dataset, we will not have to dedicate much time to prepare it for the modelling stage. With that objective, we may want to use a platform named Kaggle:

4.2.1. Kaggle

Kaggle is a data science platform which allows users to find and publish datasets, explore and build models, and enter competitions to solve data science challenges.

We are interested in finding datasets that can be useful for this project. Kaggle provides a wide variety of free datasets, which in some cases are verified by institutions. This is quite important because we need reliable data in order to build a proper medical aid system.

When we enter Kaggle, we just need to put “Alzheimer’s” in the search bar so that we can start searching a verified dataset. On the top, we have several options of searching, we have to select “Datasets” to hide all the things we are not looking for because, as previously said, Kaggle provides not only datasets but also competitions, notebooks, etc.

Once we have done this, we have a total of nine available datasets, but only four of them are verified. Out of these four, we find only two which are image datasets, so we are going to centre our attention on those. One of them consists of MRI images focusing on the hippocampus segmentation of people suffering from Alzheimer’s, and the other one consists of MRI images focusing on classifying people in four different stages of Alzheimer’s.

The second dataset adjusts more to what we were looking for, so the next step is to go deeper on the analysis of this dataset.

4.2.2. Selected dataset

The data has been hand collected from various websites with each and every label verified [21]. It consists of 6,400 MRI images in total, with 208 pixels of height and 176 pixels of width. As previously said, the images are divided into two sets (training and test sets), and they are also divided into four different classes. These classes are: *NonDemented* (3,200 images), *VeryMildDemented* (2,240 images), *MildDemented* (896 images) and *ModerateDemented* (64 images).

As we see, these labels indicate the level of severity of Alzheimer’s disease, and they almost coincide with the four first stages of Dr Barry Reisberg’s breakdown. This is exactly what we were looking for, a dataset with images of the early stages of Alzheimer’s disease.

These are four examples of the four different classes on the dataset:

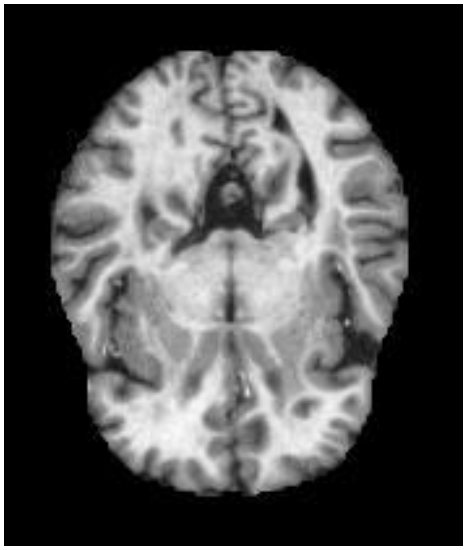


Figure 1: NonDemented

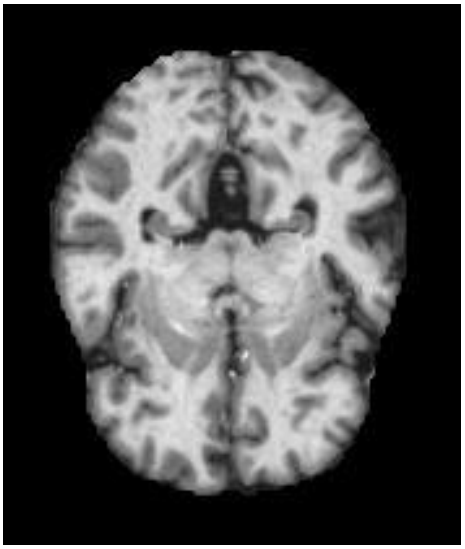


Figure 2: VeryMildDemented

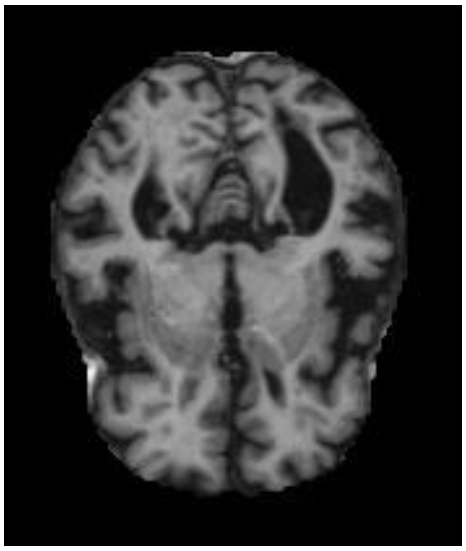


Figure 3: MildDemented

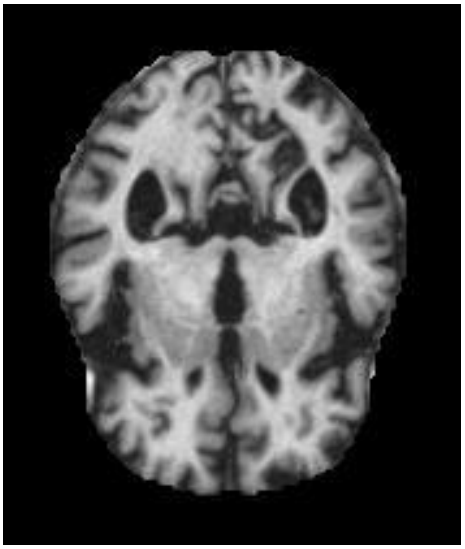


Figure 4: ModerateDemented

However, these are only 4 out of around 6,400 images. The variation of the images inside each class is high. For example, there are images in the *NonDemented* class that seem to be from another class, and this is because, due to the human eye, it is quite difficult to distinguish the difference between classes, unless you are an expert or specialist.

Even though the dataset is already split in training and test sets, we are going to join them in one unique dataset with the class division so that we can try different configurations of train and test sizes.

It is also important to remark that the original dataset (the collection of images) is not the same as the dataset we will use to train and test our models. In addition, the dataset we are going to use for training and testing is different depending on the models, so we have to distinguish between machine learning and deep learning.

4.2.2.1. Machine learning approach

Based on the initial dataset, and in order to train machine learning models, we need to process the raw data. We are working with images, so the information they give is limited, as they are just matrices of pixels. Due to this fact, we need to look for feature description techniques.

A feature descriptor is an algorithm that takes an image and outputs feature vectors. These vectors encode interesting information into a series of numbers that could be useful for training our models [22]. There are several feature descriptors such as the Histogram of Oriented Gradients (HOG), Speeded Up Robust Features (SURF) or Scale-Invariant Feature Transform (SIFT).

The feature descriptor we are going to work with is the HOG, which consists of counting the occurrences of gradient orientation in localized portions of an image [23]. This means that the HOG descriptor takes an image as input and creates a feature vector with the gradients (an increase or decrease of the pixels intensity) in its different portions. With this vector and the ones of the rest of the images, we are able to create the training and test sets. There is also another output with the purpose of visualizing what the HOG descriptor does, which is the HOG image (we are going to see an example later on in *Figure 5*, *Figure 6*, *Figure 7* and *Figure 8*).

Normally, in order to make the HOG descriptor work better, we would apply a smoothing filter and a thresholding method to the images. By doing this, the border detection might be much better because we reduce the influence of illumination effects.

However, as we are working with MRI images and there are no illumination effects (for example shades), the results are considerably better without applying these techniques.

Now, we have to decide the parameters of the HOG descriptor. Depending on the values of these parameters, the HOG image and the feature vector size change. These are three of the most important ones: *orientations* indicates the number of orientations the HOG descriptor is going to take in order to calculate the gradients; *pixels_per_cell* indicates the pixels for each cell; and *cells_per_block* indicates the number of cells for each block [24].

After searching for information [25] and thinking over, we see that, computing a formula, we can obtain the feature vector size depending on the parameters. The formula is the following:

$$row\_blocks = \left(\frac{image\_rows}{pixels\_per\_cell} - cells\_per\_block + 1\right)$$

$$col\_blocks = \left(\frac{image\_col}{pixels\_per\_cell} - cells\_per\_block + 1\right)$$

$$fv\_size = row\_blocks * col\_blocks * orientations * cells\_per\_block^2$$

The values of *pixels_per_cell* and *cells_per_block* in the HOG descriptor are tuples of two equal elements, but, in the formula, we just have to use a single value, anyone of the two.

Moreover, we can easily code this formula in Python so as to create a table with the different values of *pixels_per_cell* and *cells_per_block* calculating the sizes of the different feature vectors. We are going to set the value of *orientations* to 8 in all the cases.

The resulting table is the following:

	cells_per_block ↓					
pixels_per_cell ↓	1 x 1	2 x 2	3 x 3	4 x 4	5 x 5	6 x 6
16 x 16	1,144	3,840	7,128	10,240	12,600	13,824
8 x 8	4,576	16,800	34,560	55,936	79,200	102,816
4 x 4	18,304	70,176	151,200	257,152	384,000	527,904
2 x 2	73,216	286,752	631,584	1,098,880	1,680,000	2,366,496

Table 1: HOG parameter configurations

These feature vectors are going to build our feature dataset (*X*), which will be used to create training and test sets. This is achieved by vertically concatenating the feature vectors of all the images of the original dataset. This process is going to give us a matrix with as many rows as the number of images of the original dataset (*m*) and a number of columns equal to the size of the feature vector (*n*) we are using:

$$\begin{pmatrix} x_{00} & \cdots & x_{0(n-1)} \\ \vdots & \ddots & \vdots \\ x_{m0} & \cdots & x_{m(n-1)} \end{pmatrix}$$

We are going to generate several datasets with different values of the size of the feature vector and store them in order to test the performance later on. However, we do not have the training and test sets and the target labelling (*Y)* yet. We are not going to split and label the data until we want to test the performance because we will use different values for the number of classes and the test size, but we are going to see how it is done.

Now we are going to create the labelling of the *X*, which we are going to name *Y*. We already know the classes of all the images, so we only have to create a vector of size *m*. This vector will store a number depending on the class (0 if the image is *NonDemented*, 1 if it is *VeryMildDemented*, 2 if it is *MildDemented* and 3 if it is *ModerateDemented*).

The number of classes that we have is four, meaning the vector will store numbers going from 0 to 3:

$$(y_0 \quad \cdots \quad y_{(m-1)})$$

However, it will be interesting to try a binary classification as well (0 for *NonDemented* and 1 for the rest of the classes) to see the performance, hence the vector of labels will store only zeros and ones.

The only thing left to do is dividing *X* and *Y* so as to obtain *X_train*, *X_test*, *Y_train* and *Y_test*. Scikit-learn provides a very useful function to do this, which is called *train_test_split* [26]. We just need to input *X* and *Y*, the test size (0.1 for 10 %, 0.2 for 20 %, etc) and a random state (to avoid randomness in different executions). And the outputs are exactly *X_train*, *X_test*, *Y_train* and *Y_test*. We will try different values of the test size in order to see how it affects to the performance.

So, now that we know the feature vector sizes and the way to build the training and test sets, we have to check which of the configurations might be more interesting in order to increase the accuracy of the models, but we are going to leave this part for the tests. For now, we are going to see the difference between the first column's configurations in *Table 1* – precisely, between the HOG images. These are the four configurations with their feature vector sizes:

- *orientations* = 8; *cells_per_block* = (1, 1); *pixels_per_cell* = (16, 16)

  Size of the feature vector: 1,144

- *orientations* = 8; *cells_per_block* = (1, 1); *pixels_per_cell* = (8, 8)

  Size of the feature vector: 4,576

- *orientations* = 8; *cells_per_block* = (1, 1); *pixels_per_cell* = (4, 4)

  Size of the feature vector: 18,304

- *orientations* = 8; *cells_per_block* = (1, 1); *pixels_per_cell* = (2, 2)

  Size of the feature vector: 73,216

We are going to see the HOG images of one of the images from the original dataset as an example. In this case, we are going to see the different HOG images of the *Figure 4*, in the four configurations, because it is the worst case out of the four and it seems to be slightly more complex:

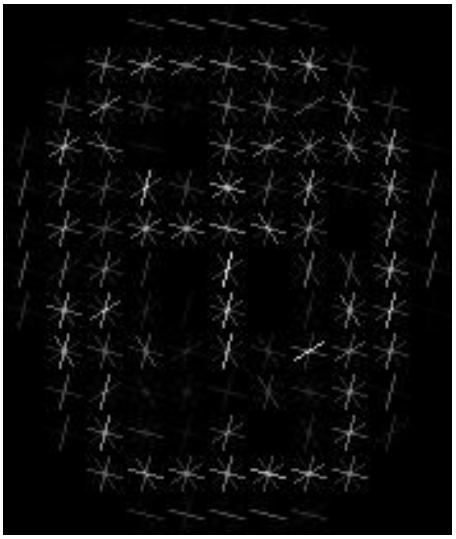


Figure 5: HOG image 1,144 feature vector size

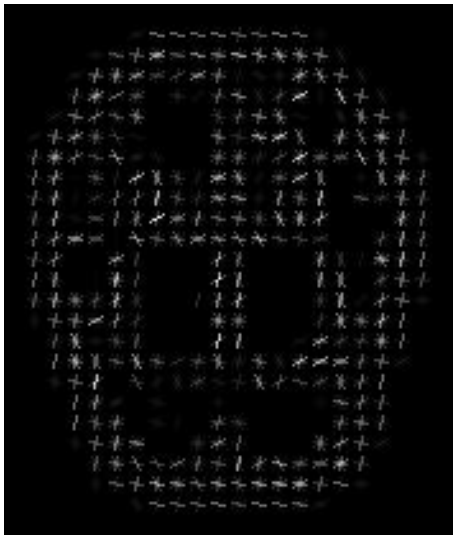


Figure 6: HOG image 4,576 feature vector size

Figure 7: HOG image 18,304 feature vector size

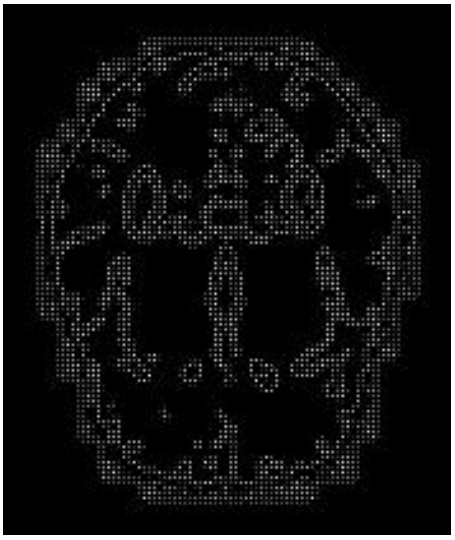


Figure 8: HOG image 73,216 feature vector size

It is quite interesting to see that whenever *pixels_per_cell* decreases, the feature vector size increases, and the level of detail on the HOG image becomes greater. This might be good in order to obtain high accuracy results in the training set. However, it could cause overfitting, which means that the model gets a high accuracy in the training set, but loses the capacity to generalize, so the results in the test set might get worse.

To sum up, now we have several *X* sets ready so as to start building machine learning models and test their performance with different parameters of both the models and the sets (number of classes and test size).

4.2.2.2. Deep learning approach

As we are going to work with convolutional neural networks, we just need to take the images to make the training and test sets. This is because the input that a CNN receives is an image.

We just need to concatenate the images of the original dataset so as to build our *X* set. Considering *m* as the number of images in the original dataset, we get a matrix with dimensions $m \times 208 \times 176$ because the images are of dimensions 208×176 as previously mentioned.

Now, it is easy to create the labelling for the *X* set (*Y*) because the images are stored in four different folders. As in the previous section, we create a vector of size *m* (number of images in the original dataset) stored with numbers (0 for *NonDemented*, 1 for *VeryMildDemented*, 2 for *MildDemented* and 3 for *ModerateDemented*). We will also try a binary classification (0 for *NonDemented*, 1 for the rest of the classes).

However, in order to train our CNNs, *Y* needs to be in one-hot encoding (a vector with the same number of columns as classes, filled with 0s and a 1 in the class column), so we need to transform the *m* size vector into a $m \times 4$ matrix if the number of classes is four, resulting in a matrix of ceros and ones like this:

$$\begin{pmatrix} y_{00} & \cdots & y_{03} \\ \vdots & \ddots & \vdots \\ y_{m0} & \cdots & y_{m3} \end{pmatrix}$$

If we use the binary classification, the one-hot encoding matrix for the labelling will only have 2 columns, one for each class.

The only thing left to do is dividing *X* and *Y* so as to get both training and test sets. As in the previous section, we can use Scikit-learn and, precisely, the function *train_test_split*, which takes *X*, *Y*, the test size and a random state and outputs *X_train*, *X_test*, *Y_train* and *Y_test*.

The process of labelling and splitting will be done later on because we want to try different test sizes and both the binary and the four classes classification.

Summarizing, now we have the *X* set ready so that we can start building our deep learning models and test their performance with different parameters of the models and the sets (number of classes and test size).

From now on, we will have to create some models and begin with the first performance tests.

4.3. First tests

Once we have the datasets (*X*) that we are going to use, we can start making the first tests in order to see which models could work better. Our objective is to reach the highest accuracy both in the training and the test sets. If we get a high accuracy in the training set, it means that our model performs right with the training data, but we want the model to generalize well with new data, and that is why we want to get high accuracy in the test set.

As seen in the previous section, we have datasets to apply both machine learning and deep learning techniques.

4.3.1. Machine learning models

In the case of machine learning, we have already built different datasets depending on the feature vector size, so we are going to try some of them in order to see how this affects the performance.

There are several types of machine learning algorithms, but, in this case, we are going to centre our attention on supervised learning algorithms. We can distinguish

in turn between two types of supervised learning algorithms – classification and regression, but we will focus on classification. This type of algorithms uses labelled data to infer a learning algorithm. The dataset is used as the basis for predicting the classification of unlabelled data through the use of these algorithms [27].

The first thing to do is deciding which classifiers we are going to use in this problem. There are several types of classifiers, and it is not possible to conclude which of them is superior because it will depend on the application and available data. We are going to select and try some of them so as to see which one is better for our datasets.

4.3.1.1. Logistic regression

A logistic regression is a machine learning algorithm which is used for classification problems. It takes an input value *X*, which is a vector of variables (in this case the feature vector) in order to return an output. This output uses the logistic sigmoid function to return a probability value, which defines the label *Y*. In [28] it is explained how the *X* goes through some calculations with the weights vector so as to get the output *Y*.

We are going to try the performance of the logistic regression with the parameters by default.

In addition, we are going to use four datasets (*X* and *Y*) out of the ones generated in the previous section (first column of *Table 1*) with the different feature vector sizes (*feat_vec_size*). We set the number of classes (*n_of_classes*) to 4 and the test size (*test_size*) to 0.3 (30 %) to generate *X_train*, *X_test*, *Y_train* and *Y_test*.

feat_vec_size	*n_of_classes*	*test_size*	*train_accuracy*	*test_accuracy*	*time*
1,144	4	30 %	74.60 %	66.10 %	1.11 s
4,576	4	30 %	99.00 %	81.90 %	3.15 s
18,304	4	30 %	100.00 %	93.10 %	13.05 s
73,216	4	30 %	100.00 %	94.40 %	62.31 s

Table 2: Logistic regression first tests

In the optimization stage, we will try different parameters for the model, the training and test sets generation, and datasets with different values of the feature vector size.

4.3.1.2. Multinomial Naive Bayes

Multinomial Naive Bayes is a machine learning algorithm which is used for classification problems, specifically, in text classification problems. This algorithm is easy to implement and computationally very efficient. Moreover, it is based on Bayes theorem, which describes the probability of an event, based on prior knowledge of conditions that might be related to the event [29].

We are going to try this model, although it is used for completely different problems, so as to compare its performance. The process of testing it is going to be the same as with the logistic regression, that is to say, with the default parameters:

feat_vec_size	***n_of_classes***	***test_size***	***train_accuracy***	***test_accuracy***	***time***
1,144	4	30 %	50.70 %	51.60 %	0.02 s
4,576	4	30 %	62.10 %	60.50 %	0.06 s
18,304	4	30 %	67.70 %	63.90 %	0.27 s
73,216	4	30 %	78.50 %	67.60 %	0.98 s

Table 3: Multinomial Naive Bayes first tests

As we can see, the execution time is significantly lower than in the logistic regression, even though the results are worse. We already knew that this algorithm is meant to be used for text classification, so there is no surprise in both the training and test sets accuracies. In addition, in spite of the feature vector size increase, the performance does not improve much, which indicates that it will not get better with bigger feature vector sizes.

4.3.1.3. SVMs

Support Vector Machines (SVMs) are as set of supervised machine learning algorithms capable of performing classification (SVC), regression and outlier detection. The linear SVM classifier works by drawing a line between two classes. All the data points that fall on one side of the line will be labelled as the first class; and all the points that fall on the other side will be labelled as the second. It does not only select a line that separates the two classes, but it also stays as far away as possible from the closest samples of both [30].

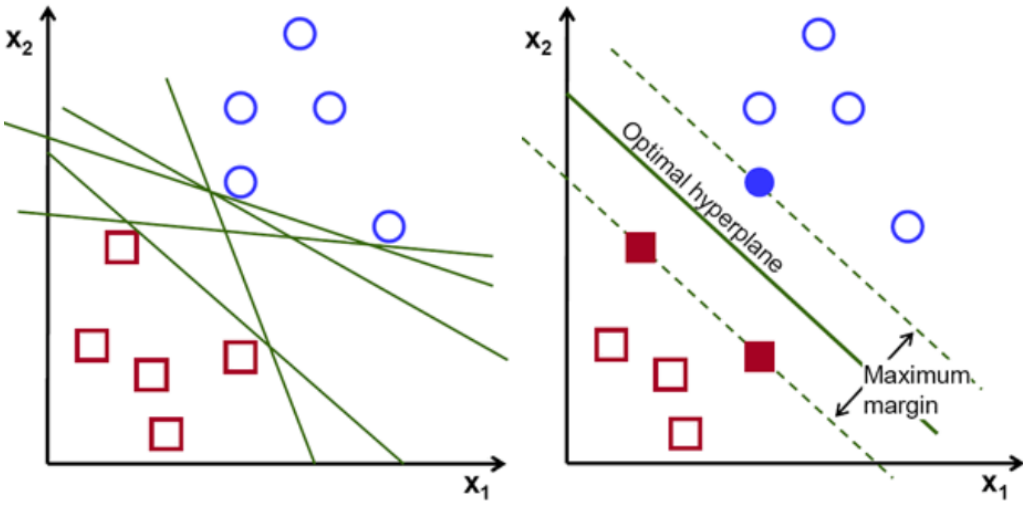


Figure 9: Linear SVM

This figure shows what a linear SVM does: it selects a line from all the possible lines between the two classes, and it creates a hyperplane which has two support vectors (discontinuous lines). As we see, if the number of features is two (*x1, x2*), we can represent it graphically. However, if the number of features is higher than three, we would not be able to represent it.

Furthermore, SVMs are capable of performing multi-class classification using different mathematical formulations.

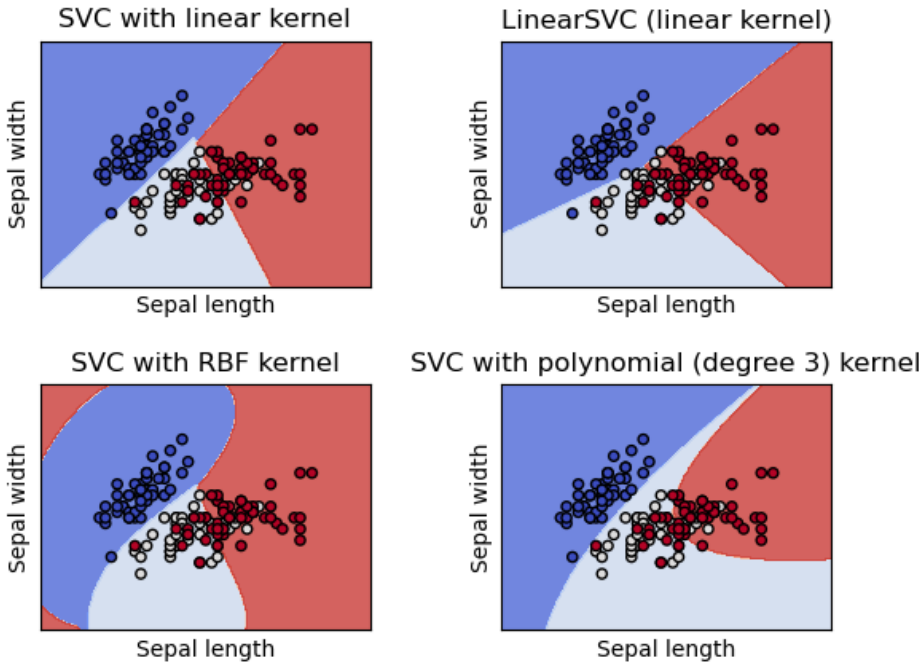


Figure 10: Multi-class SVM

Source: Scikit-learn

In order to test the performance, we are going to follow the same process as with the other models. In this case, we are also going to use all the default parameters:

feat_vec_size	*n_of_classes*	*test_size*	*train_accuracy*	*test_accuracy*	*time*
1,144	4	30 %	64.00 %	61.10 %	35.43 s
4,576	4	30 %	86.00 %	76.60 %	130.87 s
18,304	4	30 %	98.90 %	87.60 %	596.86 s
73,216	4	30 %	100.00 %	81.60 %	4,015.35 s

Table 4: SVM first tests

4.3.2. Deep learning models

As previously mentioned in *Introduction*, there are several types of artificial neural networks (ANNs), but we are going to centre our attention on convolutional neural networks (CNNs). We already know that CNNs receive an image as an input, and outputs its class.

The images are stored in the *X* set previously built and the classes in the *Y* set. Now we have to split these into training and test sets in order to train and test the CNNs.

However, we do not have a CNN built yet, so we have to build several of them in order to observe their performance. But, before that, we should know how a CNN is built. We have already seen that a CNN is a class of deep neural networks, and it is formed by a consecution of layers. We can divide these layers into three main types: convolutional layers (CONV), pooling layers (POOL) and fully connected layers (FC).

Convolutional layers employ a mathematical operation called "convolution". This operation is a simple element-wise multiplication of a filter to an input image. It is important to highlight that the convolution can be one-dimensional, two-dimensional or three-dimensional, but, in this case, we are going to use the two-dimensional. Considering we have a filter which size is $f \times f$, the multiplication is done for an $f \times f$ slice of the input image, resulting in a single value [31]. For example, if the slice for the image is the matrix *a* and the filter is the matrix *b*, the convolution operation looks like this:

$$\begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix} \times \begin{pmatrix} b_0 & b_1 \\ b_2 & b_3 \end{pmatrix} = a_0 \times b_0 + a_1 \times b_1 + a_2 \times b_2 + a_3 \times b_3 = c_0$$

So, if the image is bigger than the filter, this operation is repeated while going all over the image, resulting in an output image. This output image is called a "feature map". For example, if the input image is the matrix *a* and the filter is the matrix *b*, the convolution operation looks like this:

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{pmatrix} \times \begin{pmatrix} b_0 & b_1 \\ b_2 & b_3 \end{pmatrix} = \begin{pmatrix} c_0 & c_1 \\ c_2 & c_3 \end{pmatrix}$$

It is also important to remark that there are two more parameters apart from the filter size (*f*), the stride (*s*) and the padding (*p*). The stride indicates the number of pixels the filter is going to skip to compute the next convolution operation, and the padding is the number of extra pixels we add to the outside of the input image.

It is easy to compute the dimensions of the output image after applying the convolution to an $n \times n$ input image with this formula:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

For example, if we apply a convolution to our 208 × 176 image with 0 padding, a filter size of 3 × 3 and a stride of 1, the output image is of dimensions 206 × 174:

$$\left\lfloor \frac{208 + 2 \times 0 - 3}{1} + 1 \right\rfloor = 206 \quad \left\lfloor \frac{176 + 2 \times 0 - 3}{1} + 1 \right\rfloor = 174$$

Once the feature map is created, we can pass each value in it through a nonlinearity, such as a Rectified Linear Unit (ReLU), which is a linear activation function.

It is interesting to highlight that in most cases we will apply more than one filter to the input image. This will generate one feature map for each filter, so the output will be three-dimensional. In addition, the input can be three-dimensional as well. This is an example of a convolutional layer which receives a 208 × 176 image, and applies 32 filters of 3 × 3:

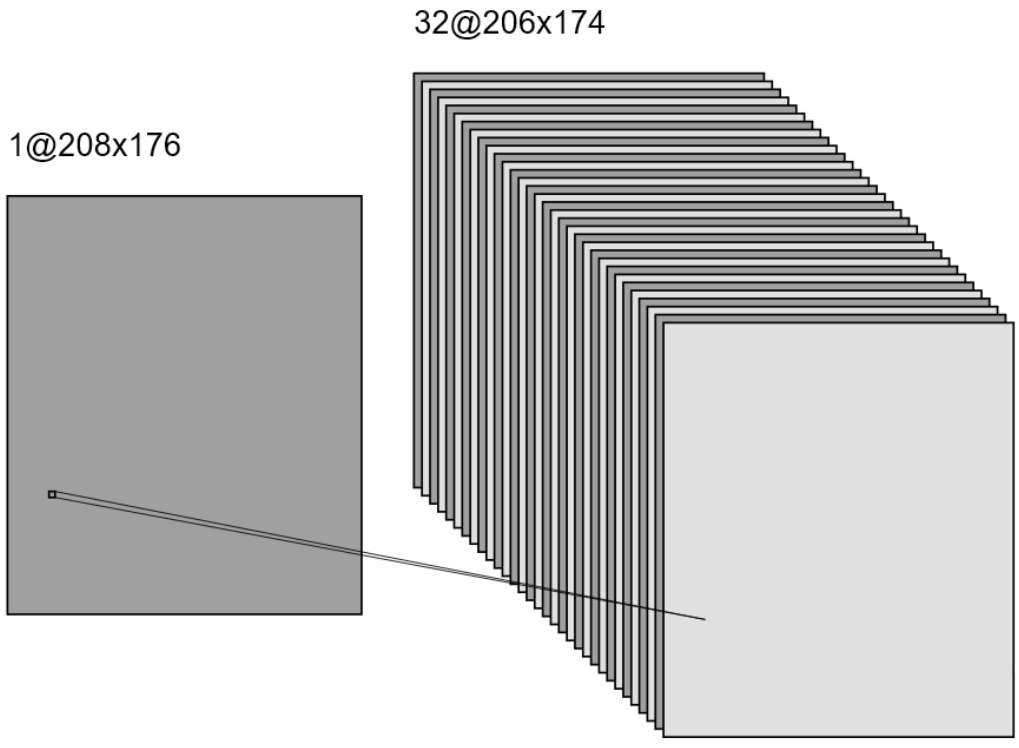


Figure 11: Convolutional layer

The output is a 206 × 176 × 32 matrix that contains the 32 feature maps, one for each filter.

Pooling layers are used to reduce the size of the representation, to speed the computation, and to make the feature detection more robust. They are layers that are added after a convolutional layer, specifically after the nonlinearity (mentioned above) has been applied to the feature maps output by the convolutional layer [32].

This type of layer involves selecting a pooling operator, which is similar to a filter that is going to be applied to a feature map. The pooling operator's dimension is normally 2 × 2 and it is applied with a stride of 2. This means that the pooling layer will always reduce the size of each feature map by a factor of 2:

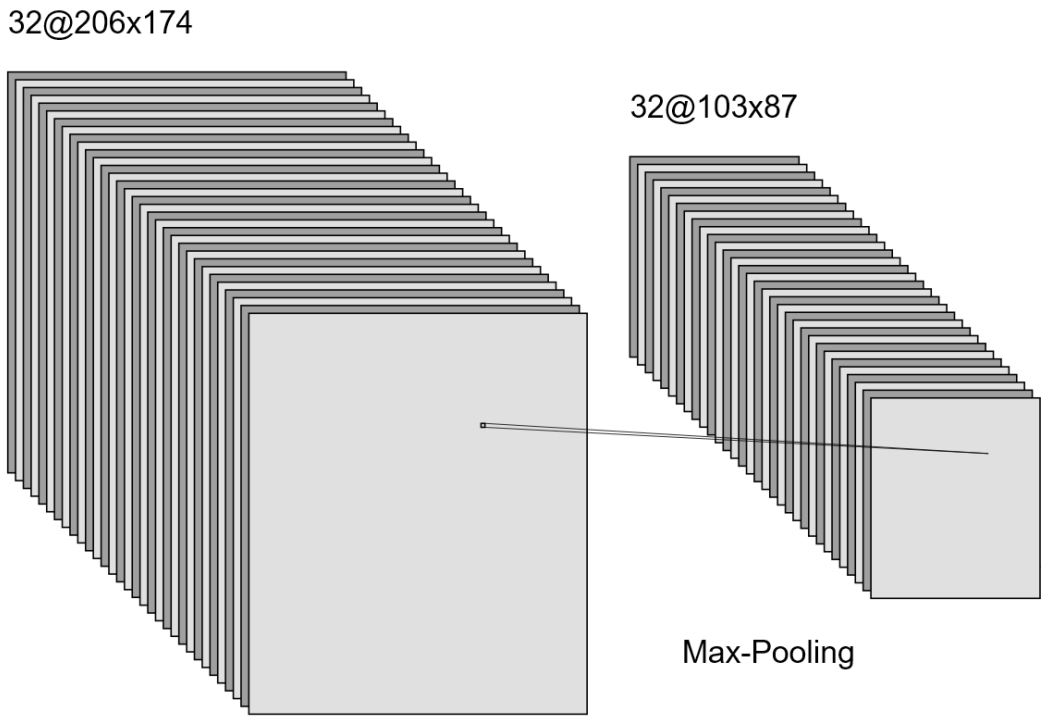


Figure 12: Pooling layer

Two of the most common ones are average pooling and maximum pooling. In this case, we will use the maximum pooling. We are going to try and apply the maximum pooling to a 4 × 4 image:

$$MaxPooling\left(\begin{pmatrix}1 & 5 & 2 & 4\\ 4 & 7 & 7 & 8\\ 2 & 3 & 9 & 6\\ 1 & 4 & 5 & 3\end{pmatrix}\right) = \begin{pmatrix}7 & 8\\ 4 & 9\end{pmatrix}$$

A fully connected layer (Dense in Keras) is where all its units are connected to all the units of the previous layer. We get the output of the last convolutional or pooling layer and we flatten it, this means unrolling all its values into a vector. This vector is going to be the input of a fully connected layer [33].

The output that we wanted was the image's class, and the only thing that we have is an input fully connected layer built by a vector with a lot of feature numbers. Therefore, we need an output fully connected layer with a SoftMax activation and a dimension equal to the number of classes.

The SoftMax function receives as input a vector of K real numbers and normalizes it into a probability distribution of K probabilities. After applying this function, each component will be in the interval (0,1), and the components will add up to 1. This is used for multi-classification, in order to decide the class of the input image taking into account the highest probability [34].

This figure shows the flatten and both of the parts of the fully connected layer (input and output):

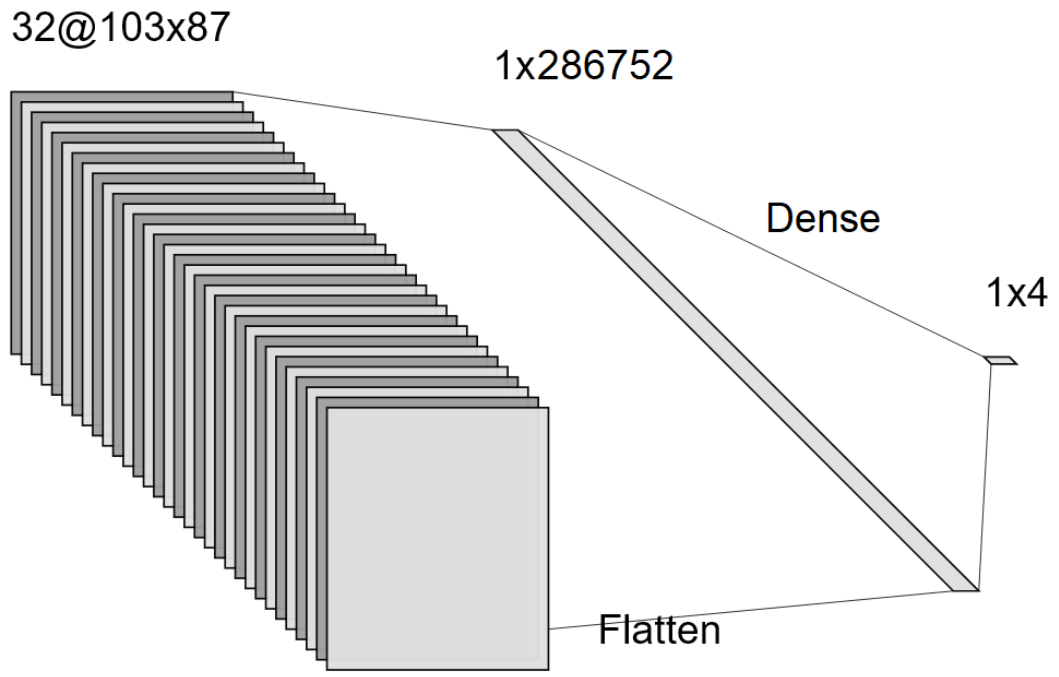


Figure 13: Fully connected layer

Having understood the basic operational of a CNN, we can now build some models.

4.3.2.1. First model

This model is a sequence of three layers, and it has a total of 4,510,372 trainable parameters:

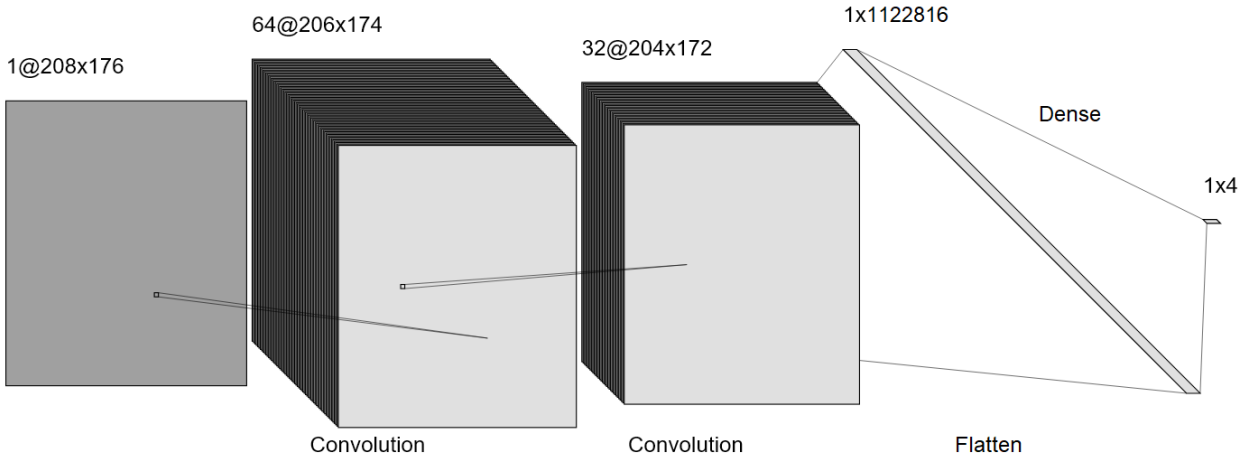


Figure 14: First model

The precise architecture is this one:

- Convolutional layer (Conv2D) with an *input_shape* of (208, 176, 1), a ReLU activation, and 64 filters with *kernel_size* set to 3.
- Convolutional layer (Conv2D) with a ReLU activation, and 32 filters with *kernel_size* set to 3.
- Input fully connected layer with 1,122,816 units.
- Output fully connected layer with 4 units and a SoftMax activation.

For these first tests, we are going to set the number of classes to 4 and the test size to 0.3 (30 %) to generate the *X_train*, *X_test*, *Y_train* and *Y_test* from our *X* and *Y* generated in *Deep learning approach*.

These are the results of the first model:

n_of_classes	***test_size***	***train_accuracy***	***test_accuracy***	***time***
4	30 %	100.00 %	98.44 %	12,150.87 s

Table 5: First test of the first model

4.3.2.2. Second model

The second model we are going to build is slightly more complex than the first one. It has a total of six layers, and 1,064,452 trainable parameters:

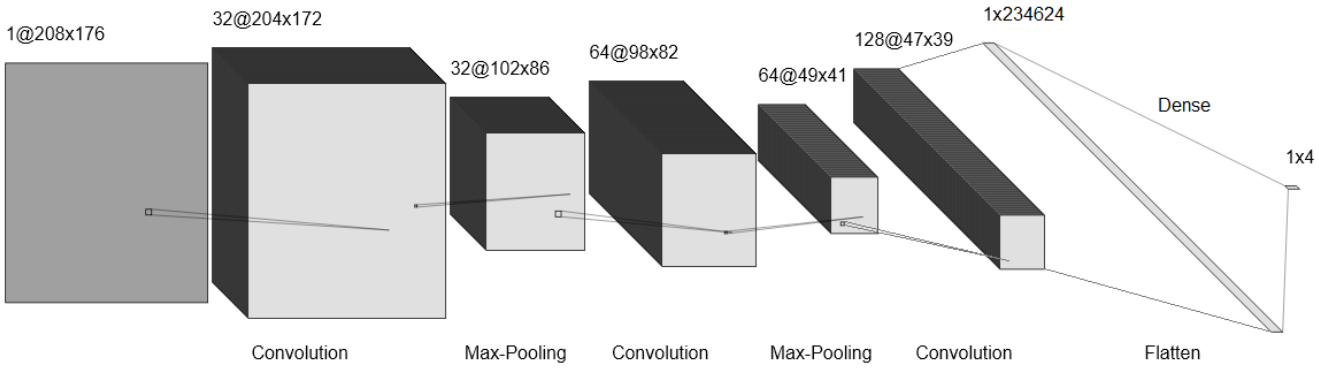


Figure 15: Second model

It has a more varied architecture because now we are using pooling layers apart from convolutional and fully connected layers. This affects to the number of parameters the CNN has to train. In this model, the number of parameters is less than in the previous one because the pooling layers reduce considerably the dimensions of its inputs. These are the layers with their variables' values:

- Convolutional layer (Conv2D) with an *input_shape* of (208, 176, 1), a ReLU activation, and 32 filters with *kernel_size* set to 5.
- Max-Pooling layer (MaxPooling2D) with a *pool_size* and stride of 2.
- Convolutional layer (Conv2D) with a ReLU activation, and 64 filters with *kernel_size* set to 5.
- Max-Pooling layer (MaxPooling2D) with a *pool_size* and stride of 2.
- Convolutional layer (Conv2D) with a ReLU activation, and 128 filters with *kernel_size* set to 3.
- Input fully connected layer with 234,624 units.
- Output fully connected layer with 4 units and a SoftMax activation.

We are going to follow the same method as with the first model in order to test the performance. These are the results of the second model:

n_of_classes	***test_size***	***train_accuracy***	***test_accuracy***	***time***
4	30 %	100.00 %	98.02 %	3,135.96 s

Table 6: First test of the second model

4.3.2.3. Third model

The third model has a very similar architecture to the second one, it is built with a total of seven layers. The only thing that changes, regarding the second model apart from the extra layer, is the number and dimensions of the filters in the convolutional layers. That is why this one has 146,180 trainable parameters:

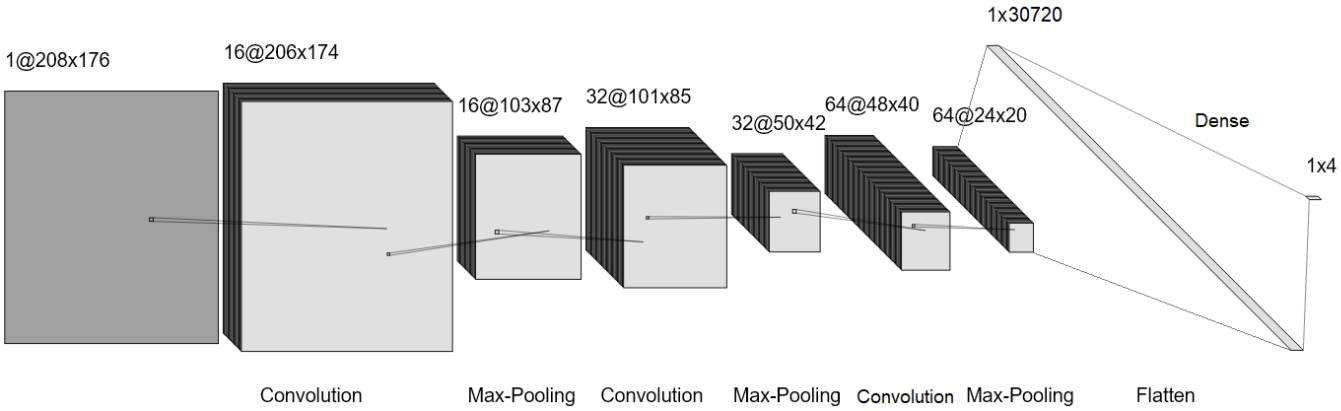


Figure 16: Third model

And these are the layers with all the values of the variables:

- Convolutional layer (Conv2D) with an *input_shape* of (208, 176, 1), a ReLU activation, and 16 filters with *kernel_size* set to 3.
- Max-Pooling layer (MaxPooling2D) with a *pool_size* and stride of 2.
- Convolutional layer (Conv2D) with a ReLU activation, and 32 filters with *kernel_size* set to 3.
- Max-Pooling layer (MaxPooling2D) with a *pool_size* and stride of 2.
- Convolutional layer (Conv2D) with a ReLU activation, and 64 filters with *kernel_size* set to 3.
- Max-Pooling layer (MaxPooling2D) with a *pool_size* and stride of 2.
- Input fully connected layer with 30,720 units.
- Output fully connected layer with 4 units and a SoftMax activation.

These are the results for the third model:

n_of_classes	***test_size***	***train_accuracy***	***test_accuracy***	***time***
4	30 %	100.00 %	97.92 %	1,614.31 s

Table 7: First test of the third model

4.4. Optimization

Once we have obtained the first results of the models' performance, we can start with the optimization. In this section, we are going to try and improve the models in order to obtain better results. In other words, we are going to tune the parameters of the models and test different settings of the datasets so as to get the best configurations.

4.4.1. Machine learning

As previously seen, we have already built some models in order to evaluate their performance. The logistic regression and the SVM obtained increasingly better results as the dataset's feature vector size increased. However, the multinomial Naive Bayes reached low accuracy in both the training and test sets, therefore, we are not going to optimize it.

So as to tune the logistic regression and the SVM parameters, we are going to use GridSearchCV, which is a function provided by Scikit-learn. We just need to input an estimator (in this case the classifier) and a grid of the estimator's parameters we want to try. The function performs an exhaustive search over the specified grid of parameters, and it uses the cross-validation (a technique used to test the effectiveness of a model) in order to optimize this grid [35].

It is important to remark that we are going to use the training and test sets, whose feature vectors have a size of 18,304, for the GridSearchCV. This is because they had robust results in the first tests, and they are not as time-consuming as the training and test sets with 73,216 feature vector size. In addition, we have followed the same structure of the first tests for the GridSearchCV, using 30 % of the data for the test set and a 4-class classification.

4.4.1.1. Logistic regression optimization

For the logistic regression we are going to tune the following parameters [36]:

- *penalty* (default *l2*) is used to specify the norm used in the penalization, which is the sum of the values of the weights. In this case, we are going to use the *l2* penalty because it is the only one that supports every *solver* (another parameter).

- *C* (default *1*) is the inverse of the regularization strength. This means that the smaller this parameter is, the stronger the regularization gets. The regularization is a technique that reduces the importance of parameter values to reduce overfitting because, if our data had a lot of different variables, the model would try to adjust the parameters as much as it could and would lose the ability to generalize in new data.
- *solver* (default *lbfgs*) is the algorithm used in the cost function minimization. As we are using the *l2* penalty, we will not have any compatibility problems with the selected solver.
- *max_iter* (default *100*) is the maximum number of iterations taken for the solvers to converge.

Now that we know more about the parameters we are going to tune, we can define the grid of their different values:

- ***penalty***: *l2.*
- ***C****: 0.01, 0.1, 1, 10, 100.*
- ***solver****: newton-cg, lbfgs, liblinear.*
- ***max_iter****: 100, 200.*

The next step is executing the GridSearchCV with the parameter grid above. We obtained that the best possible parameters for the logistic regression, using the training and test sets with 18,304 feature vector size, are the following:

- ***penalty****: l2;* ***C****: 10;* ***solver****: lbfgs;* ***max_iter****: 100.*

We can see that the best configuration is almost the default one, except the *C*, which is 10. The higher the *C* is, the weaker the regularization. This means that the model does not regularize much with this dataset. These parameters are the ones that we are going to use in order to improve the performance of our logistic regression.

4.4.1.2. SVM optimization

For the SVM we are going to tune the following parameters [37]:

- *C* (default *1*) is the same as in the logistic regression.

- *gamma* (default *scale*) defines how far the influence of a single training example reaches, this means that if *gamma* is small, the model does not fit the data, whereas if *gamma* is big, the model tends to overfit. We can clearly see this in a 2-variable dataset:

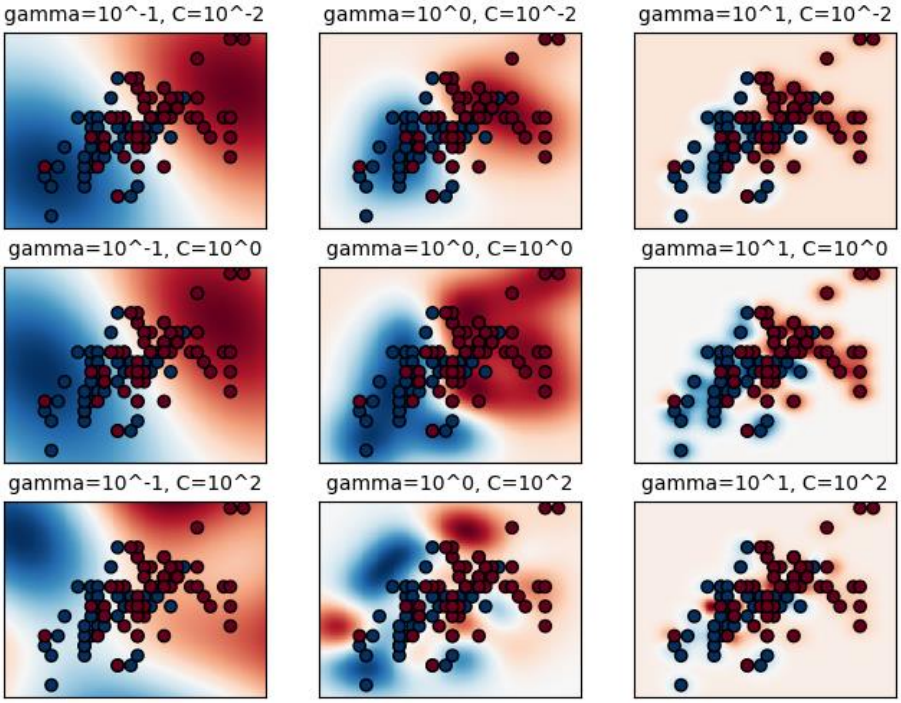


Figure 17: SVM gamma parameter Source: Scikit-learn

The default value (*scale*) means that *gamma* takes the value computed by the following formula:

$$\frac{1}{(n\_features * X.var())}$$

Where *n_features* is the number of columns of our dataset (features) and *X.var()* is the variance of the matrix *X*.

- *kernel* (default *rbf*) is, in broad strokes, the function that takes data as input and transforms it into the required form of processing data. Sometimes we cannot classify data with a linear decision boundary. Therefore, we use this *kernel* function, which transforms the data so that a non-linear decision surface is able to be converted into a linear equation in a higher number of dimension spaces.

We now know how these parameters affect the SVM, so we can build the grid that we will use in the GridSearchCV:

- ***C**: 0.1, 1, 10, 100.*
- ***gamma**: scale, 0.001, 0.01, 0.1.*
- ***kernel**: linear, rbf, sigmoid, poly.*

Finally, we execute the GridSearchCV with this grid and we obtain that the best parameters for the SVM, using the dataset with 18,304 feature vector size, are:

- ***C**: 1; **gamma**: scale; **kernel**: poly.*

We can see that the best kernel function is *poly*. This means that the SVM uses a polynomial kernel. Now that we know this, there is an additional parameter (*degree*, ignored by the other kernel functions) that indicates the degree of the polynomials. The default value for *degree* is *3*. We are going to try different values (*2, 3, 4* and *5)* for this parameter in order to see if the performance gets better. The best configuration for the SVM after optimizing is:

- ***C**: 1; **gamma**: scale; **kernel**: poly; **degree**: 3.*

We are going to use these parameters to improve the SVM performance.

4.4.1.3. Dataset optimization

Now that we already have both the logistic regression and the SVM with their best possible parameters for this problem, the only thing left to do is check whether the performance is affected when we change the test size and the number of classes. We will also try datasets with different feature vector size, so that we can decide which of them is more viable.

In *First tests*, we tried datasets with 4 different feature vector sizes. In this stage we are going to use the optimized logistic regression and SVM to test the performance of these datasets and a few more in order to see if we can improve the performance. The feature vector sizes are these ones:

- *1,144; 4,576; 12,600; 16,800; 18,304; 34,560; 55,936; 73,216.*

We are also going to try different parameters for the test size and the number of classes:

- ***test_size**: 10 %, 20 %, 30 %.*
- ***n_of_classes**: 2, 4.*

In the next section, we will observe the performances of all of these datasets with the different configurations of the test size and the number of classes.

4.4.2. Deep learning

We have already built three convolutional neural networks in order to check the performance. In *First tests* we obtained a surprisingly high accuracy in both the training and the test sets. However, we will see whether the performance changes after modifying the dataset's test size and number of classes.

The parameters we are going to try are the following:

- ***test_size**: 10 %, 20 %, 30 %.*
- ***n_of_classes**: 2, 4.*

As with the machine learning techniques, we will observe the performance of the dataset with the different configurations of the test size and number of classes in the next section.

4.5. Results

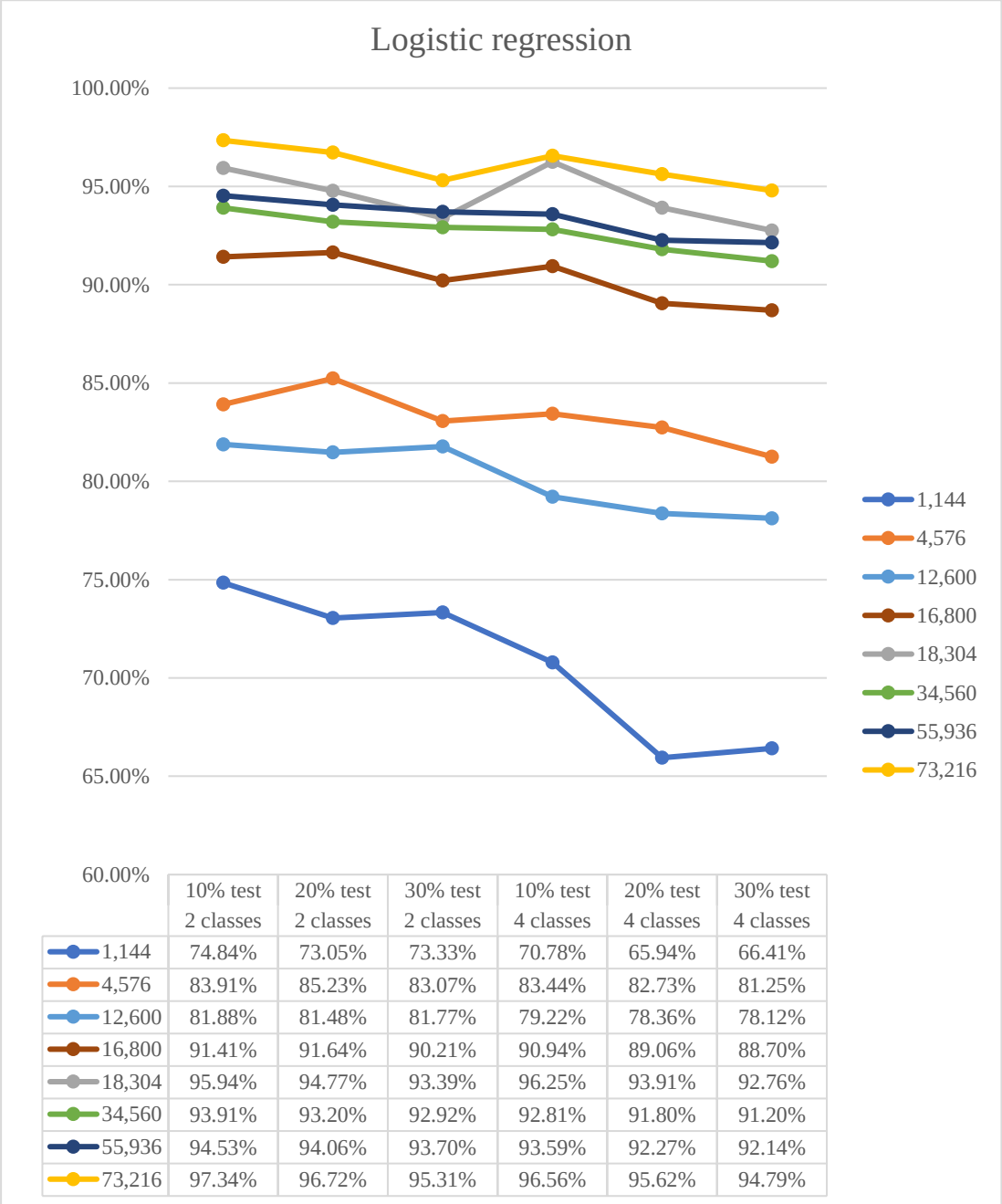
After trying all these configurations and parameters, we have generated several tables with all the results, and now we can extract knowledge from them. We are going to create graphs in order to have a general overview of these results.

4.5.1. Machine learning results

In the case of machine learning, we have already found the best possible configurations for the logistic regression and the SVM, and we have also tried different dataset tuning.

The first graph we are going to create is useful so as to compare the performance of the different datasets. We are going to call each dataset by its number of feature vector size. It is important to remark that we are not going to consider the train accuracy because its 100 % in almost all the datasets both with the logistic regression and the SVM. The vertical axis represents the accuracy obtained in the test set, and the horizontal axis shows the information of the dataset's parameters, the percentage of the dataset destined to the test set and the number of classes.

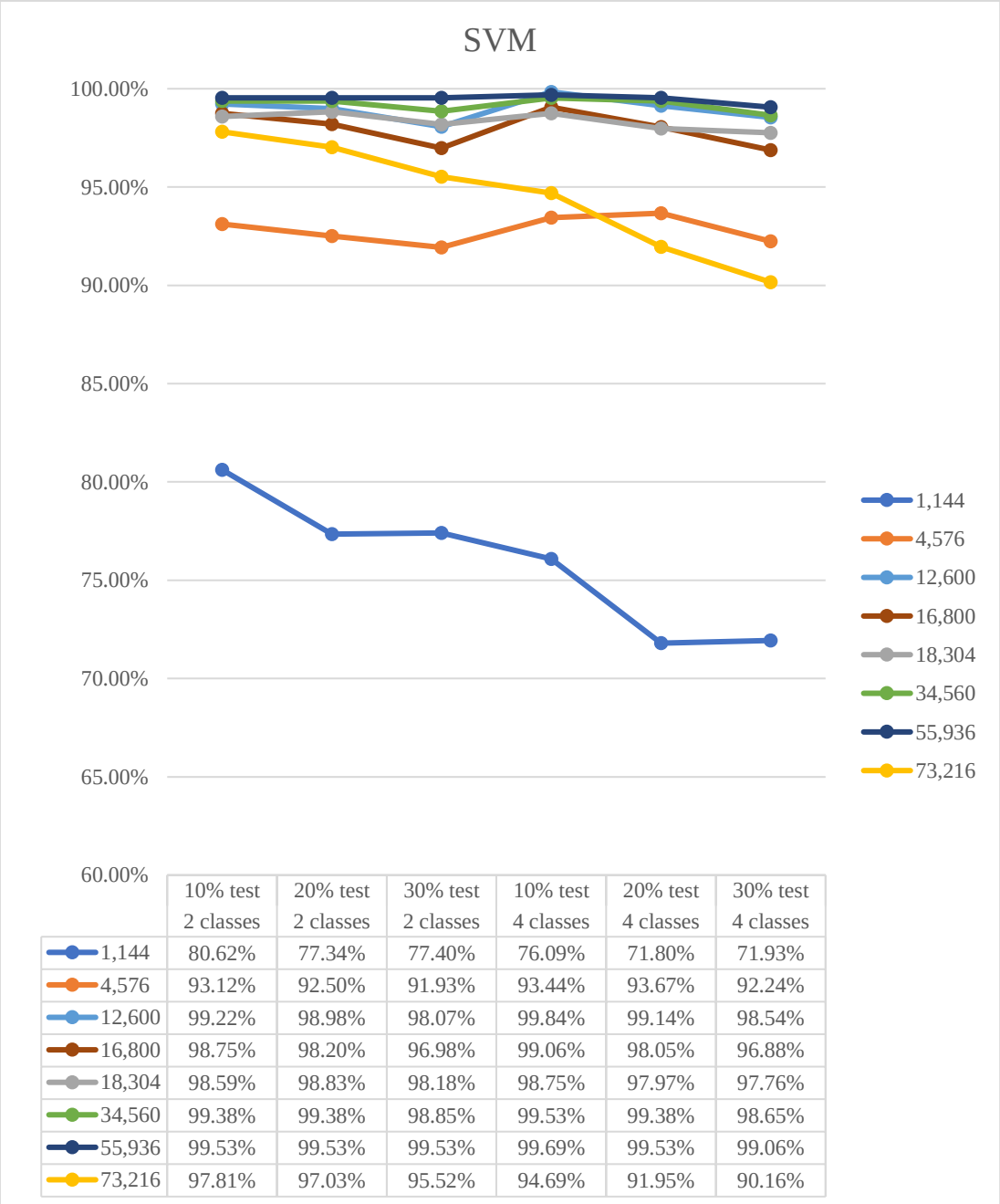
The logistic regression graph:



	10% test 2 classes	20% test 2 classes	30% test 2 classes	10% test 4 classes	20% test 4 classes	30% test 4 classes
1,144	74.84%	73.05%	73.33%	70.78%	65.94%	66.41%
4,576	83.91%	85.23%	83.07%	83.44%	82.73%	81.25%
12,600	81.88%	81.48%	81.77%	79.22%	78.36%	78.12%
16,800	91.41%	91.64%	90.21%	90.94%	89.06%	88.70%
18,304	95.94%	94.77%	93.39%	96.25%	93.91%	92.76%
34,560	93.91%	93.20%	92.92%	92.81%	91.80%	91.20%
55,936	94.53%	94.06%	93.70%	93.59%	92.27%	92.14%
73,216	97.34%	96.72%	95.31%	96.56%	95.62%	94.79%

Figure 18: Logistic regression results

The SVM graph:



	10% test 2 classes	20% test 2 classes	30% test 2 classes	10% test 4 classes	20% test 4 classes	30% test 4 classes
1,144	80.62%	77.34%	77.40%	76.09%	71.80%	71.93%
4,576	93.12%	92.50%	91.93%	93.44%	93.67%	92.24%
12,600	99.22%	98.98%	98.07%	99.84%	99.14%	98.54%
16,800	98.75%	98.20%	96.98%	99.06%	98.05%	96.88%
18,304	98.59%	98.83%	98.18%	98.75%	97.97%	97.76%
34,560	99.38%	99.38%	98.85%	99.53%	99.38%	98.65%
55,936	99.53%	99.53%	99.53%	99.69%	99.53%	99.06%
73,216	97.81%	97.03%	95.52%	94.69%	91.95%	90.16%

Figure 19: SVM results

We can see that, with the logistic regression, we generally get increasingly better results as the feature vector size of the dataset increases. However, the SVM does not follow the same rule: the performance increases until the dataset with 55,936 feature vector size, then it starts to go down.

In addition, we can clearly see that we get much better results using the SVM, reaching a 99.69 % of accuracy in the test set, which is its peak performance. This is obtained with the 55,936 feature vector size dataset in the binary classification with 10 % of the data destined to the test set.

We are specifically interested in the most difficult configuration, which is the 4-class classification with 30 % of the data destined to the test set. The logistic regression reaches a 94.73 % test accuracy with the 73,216 feature vector size dataset, whereas the SVM is capable of obtaining a 99.06 % test accuracy with the 55,936 feature vector size dataset. In order to check how these are performing, we are going to build a confusion matrix for each of them, so that we can compare their predictions.

Confusion matrix of the logistic regression

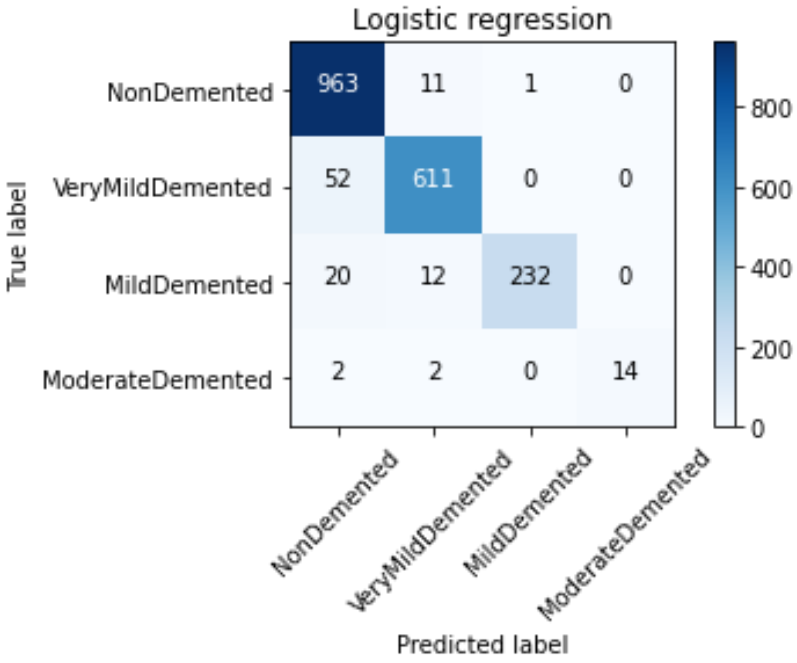


Figure 20: Logistic regression confusion matrix

Confusion matrix of the SVM:

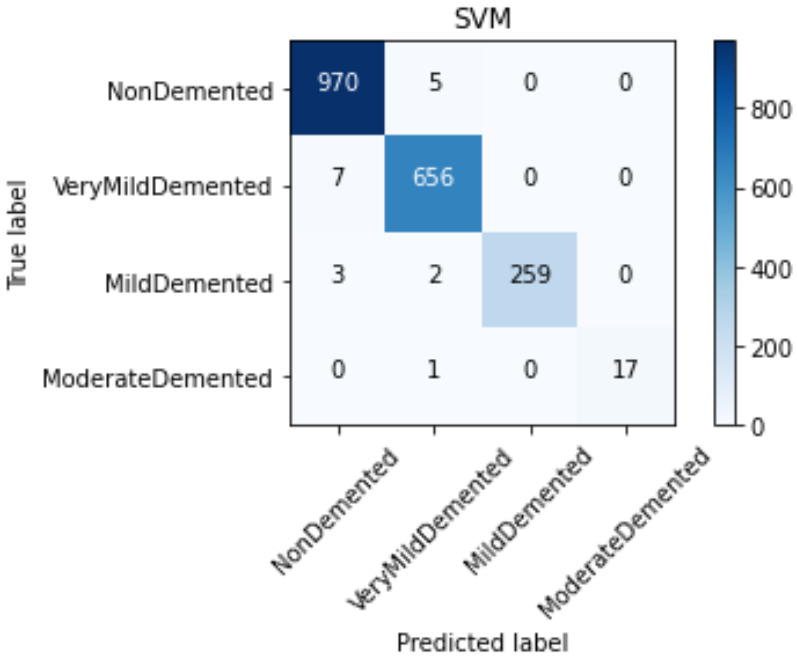


Figure 21: SVM confusion matrix

In these confusion matrices, we can clearly see the predictions of both the logistic regression and the SVM, having into account the true labels. The elements of the main diagonal are the correctly predicted cases. We can consider as the worst predictions the ones in the first column (minus the main diagonal), this is because our models are predicting that the patient does not have Alzheimer's while having it.

The SVM actually predicts correctly 1,902 cases out of 1,920 (18 wrong predictions), which as previously said means that it reaches a 99.06 % test accuracy. On the other hand, the logistic regression does much worse, predicting correctly 1,820 cases out of 1,920 (100 wrong predictions), which is a total of 94.73 % test accuracy.

In the Alzheimer's disease diagnosis, we want as much accuracy as we can, that is why we have not taken into account variables such as the training time. However, it is important to remark that the SVM training process implies much more time that the logistic regression. In this case, the SVM trains in a total of 1,582.67 seconds, while the logistic regression trains in just 71.90 seconds.

As the SVM is much more interesting for our problem, we are going to visualize some of its failed predictions:

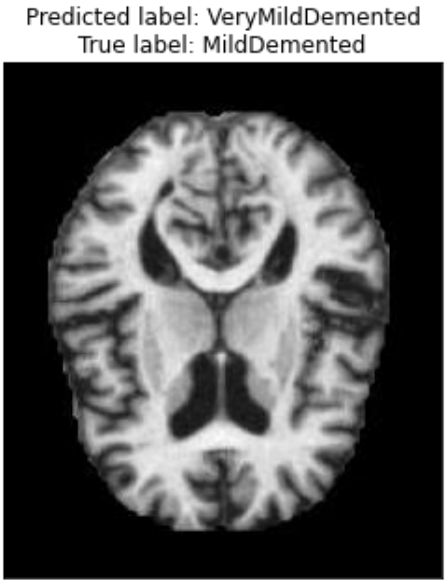


Figure 22: SVM failed prediction 1

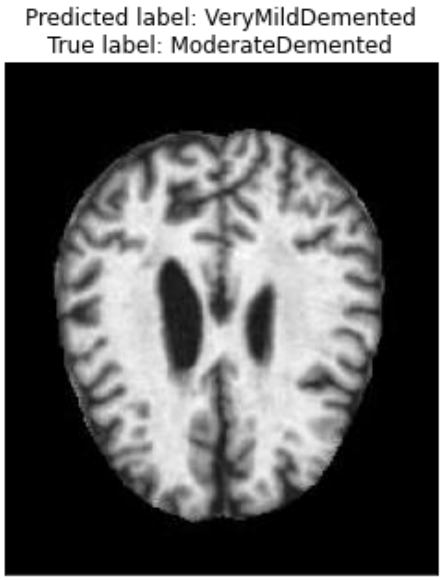


Figure 23: SVM failed prediction 2

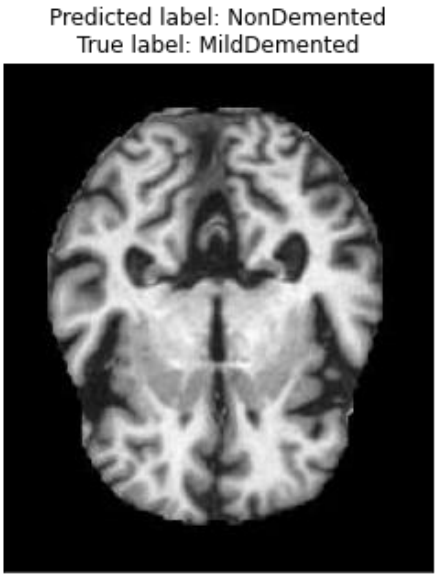


Figure 24: SVM failed prediction 3

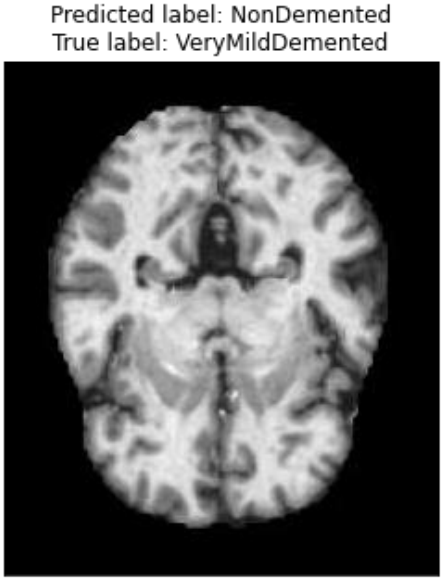


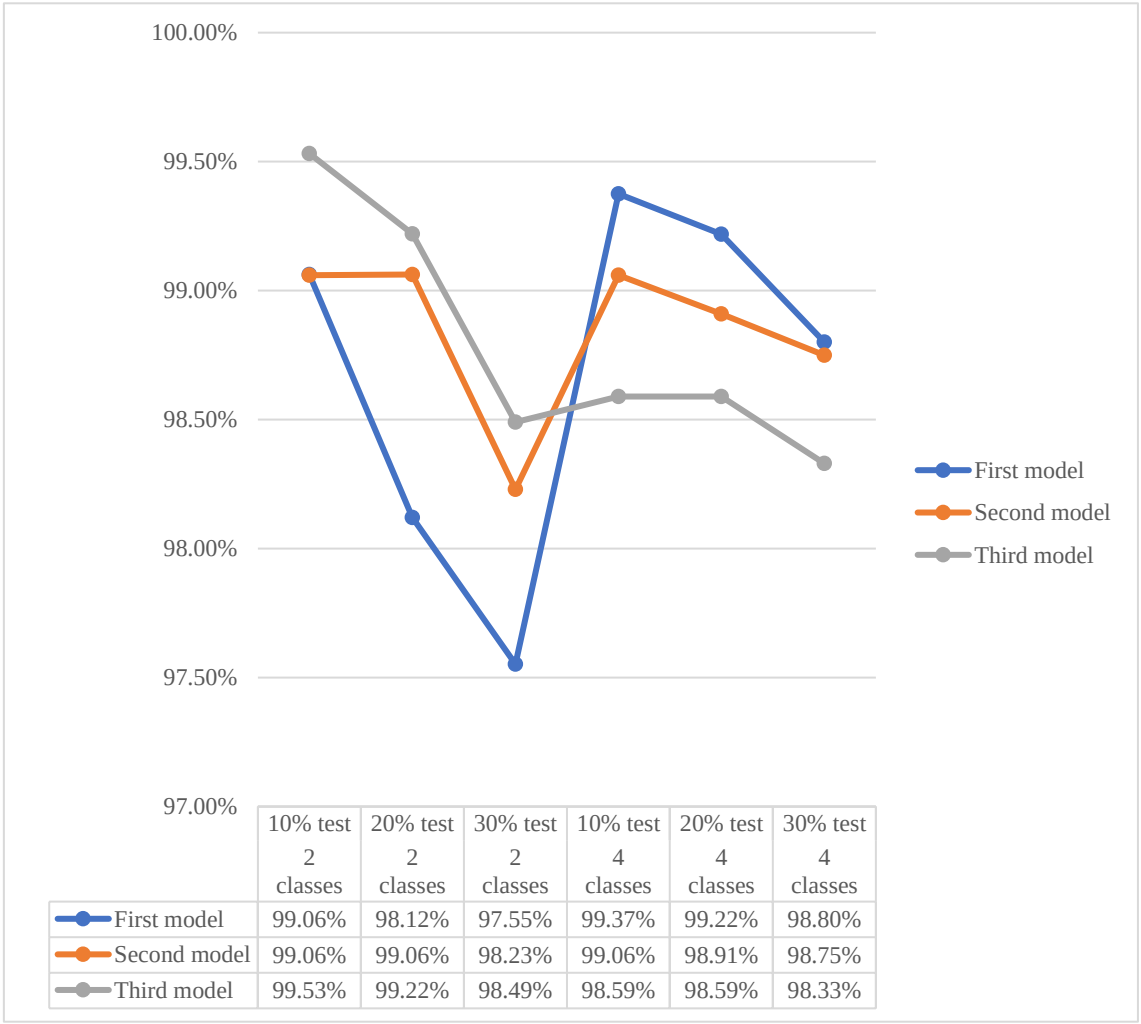
Figure 25: SVM failed prediction 4

The predictions in *Figure 22* and *Figure 23* show that the patient has Alzheimer, but not in the stage that really is. These cases are not as bad as the ones in *Figure 24* and *Figure 25* because the SVM is predicting that the patient does not have Alzheimer, but he or she has it.

Moreover, we can see that, for the human eye, it is not possible to classify these cases properly, unless you are a specialist, but the SVM predicts correctly most of them.

4.5.2. Deep learning results

For the deep learning models, we tried parameters for both the number of classes in the classification and the percentage of the data destined to the test set. As with the machine learning techniques, we are not going to consider the train accuracy because it is 100 % in all the cases. We are going to visualize the performance of the three models in the following graph, which shows the test accuracy in the vertical axis and the number of classes and test size in the horizontal axis:



	10% test 2 classes	20% test 2 classes	30% test 2 classes	10% test 4 classes	20% test 4 classes	30% test 4 classes
First model	99.06%	98.12%	97.55%	99.37%	99.22%	98.80%
Second model	99.06%	99.06%	98.23%	99.06%	98.91%	98.75%
Third model	99.53%	99.22%	98.49%	98.59%	98.59%	98.33%

Figure 26: Deep learning models results

We can see that the performance of the three models is generally decent. The third model performs better with a binary classification, whereas the first model would be the chosen if we want to work with four classes. The second model has more stable results in both the binary and the multi-class classification.

All the results are above 97.50 % test accuracy, but the peak performance is reached by the first model in the binary classification with a 10 % of the data destined to the test set and a test accuracy of 99.53 %. It is interesting to see that this accuracy decays whenever the test set is extended. This might be caused because the training set gets smaller and may not be big enough to properly train the convolutional neural networks.

In *First tests*, the performance of the convolutional neural networks was superior in all the cases. However, after optimizing the SVM, their performance is slightly below.

We are going to inquire in the most difficult configuration, as with the machine learning techniques, which is the 4-class classification, using a 30 % of the data for the test set. With the deep learning techniques, the model that obtains the highest accuracy is the first one, with a 98.80 % of correct predictions. This means that it reaches a 0.26 % less accuracy than the SVM with the same configuration. We are going to build its confusion matrix so as to compare them [38]:

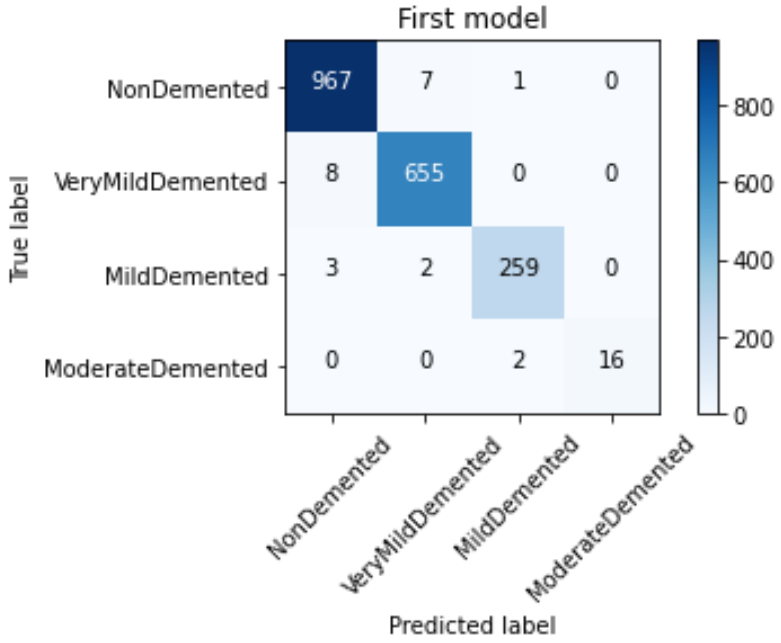


Figure 27: Confusion matrix of the first model

We can appreciate that the confusion matrix of the first model and the SVM are quite similar. The first model predicts correctly 1,897 cases out of the 1,920 (23 wrong predictions).

The training process of the first model with this configuration has lasted a total of 17,851.05 seconds. The first model training times have been higher than the second and third models' ones, and this is because the number of trainable parameters is much bigger. We can see the accuracy and loss progress through these graphs (accuracy's value 1 equals to 100 %):

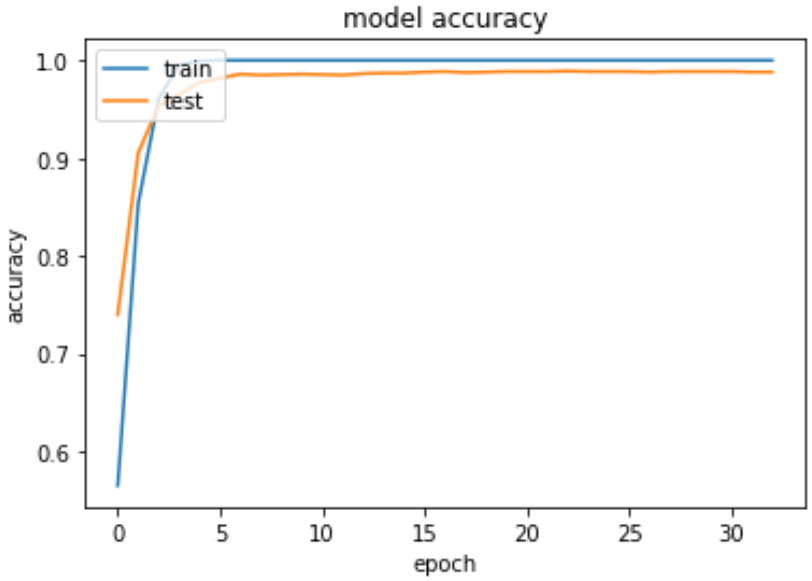


Figure 28: Accuracy progress of the first model

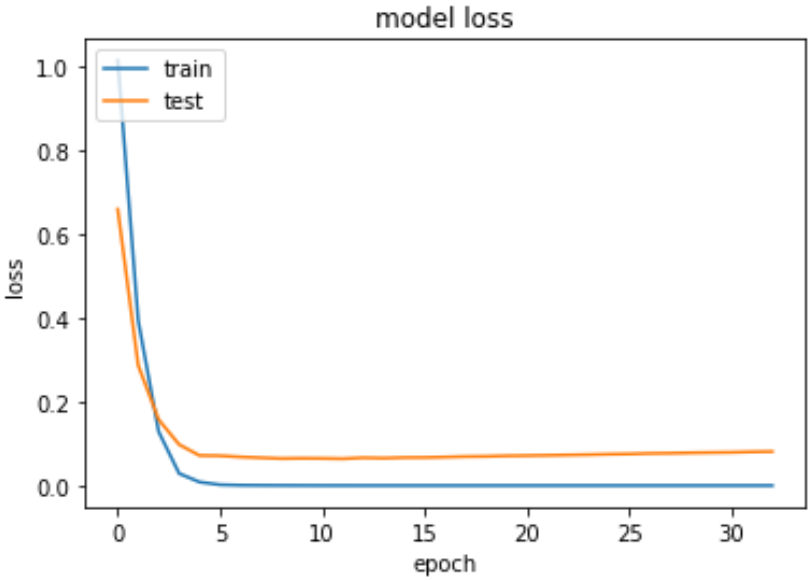


Figure 29: Loss progress of the first model

We can see that the training process stops in the 32nd epoch, an epoch refers to one cycle through the full training set. Normally, a convolutional neural network needs a few epochs to train. However, in this case it needed 32 due to the fact that we added an early stopping method with the *patience* value established to 10. This means that the training process is not going to stop until the test accuracy does not improve in 10 epochs. We have done this in order to obtain the highest possible test accuracy. With smaller values of *patience*, the results were not good enough.

Now, we are going to visualize which are the outputs of the convolutional layers in the first model. As previously seen, the first model has two convolutional layers. We are interested in the filters of these layers and what do they output (feature maps). There are a lot of filters, but we are going to take one in order to see how it works [39]:

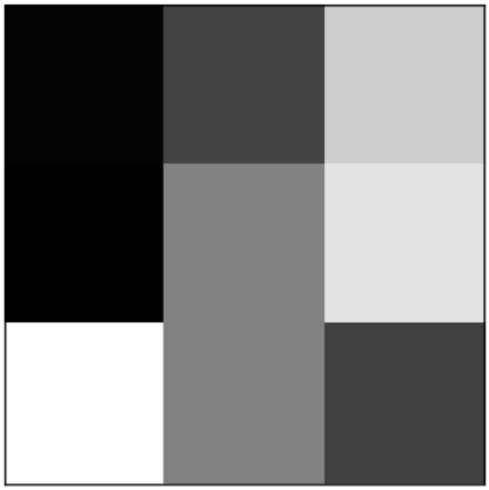


Figure 30: Filter example

$$\begin{pmatrix} -0.1192 & -0.0536 & 0.0818 \\ -0.1192 & 0.0067 & 0.1023 \\ 0.1294 & 0.0063 & -0.0546 \end{pmatrix}$$

This filter is one of the 64 filters in the first convolutional layer. We have normalized its values (matrix below *Figure 30*) in order to represent it graphically. This convolutional layer takes an image as input and it uses the 64 filters in order to output 64 different feature maps.

We are going to see which is the outputted feature map after applying this filter to the following image:

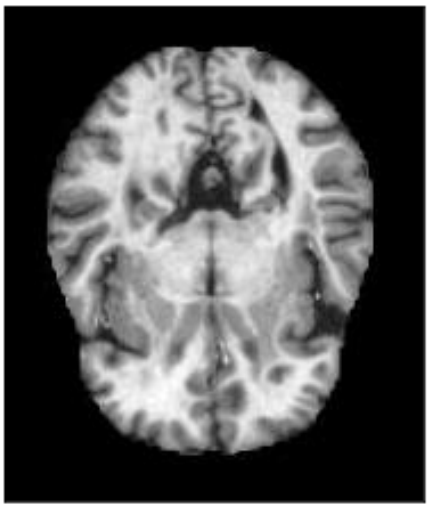


Figure 31: Input image example

Figure 32: Output feature map example

As we see, if we apply the filter to the image in *Figure 31*, we get the feature map in *Figure 32* and this feature map will be one of the inputs for the next layer. Now, we could visualize one of the feature maps outputted in the last layer before the fully connected layers in order to see which features uses the first model to classify. One of these feature maps is the following:

Figure 33: Last layer output feature map example

The white pixels in this feature map and the other ones, outputted in the last convolutional layer, are the features that the model uses in order to predict the class of the image (after passing through the fully connected layers).

Finally, we are going to visualize some of the failed predictions of the first model:

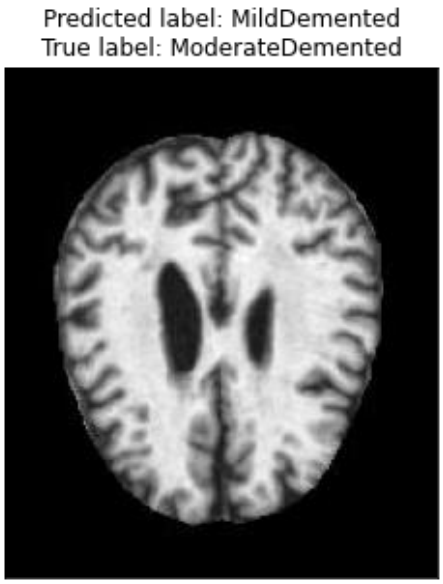


Figure 34: Failed prediction 1 of the first model

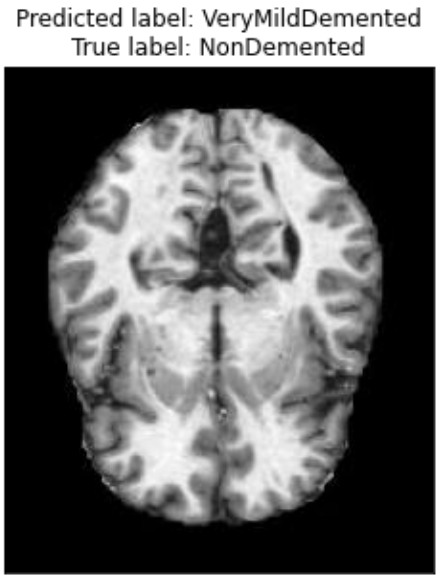


Figure 35: Failed prediction 2 of the first model

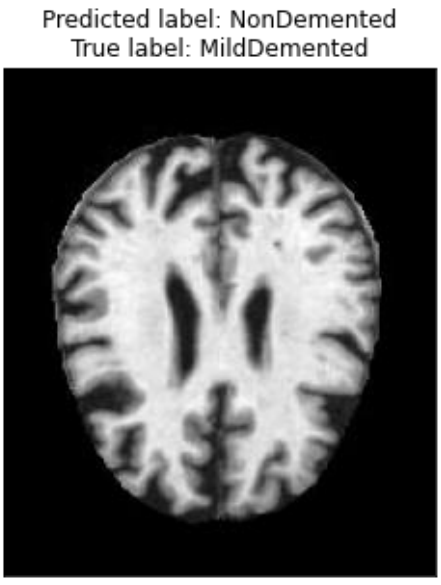


Figure 36: Failed prediction 3 of the first model

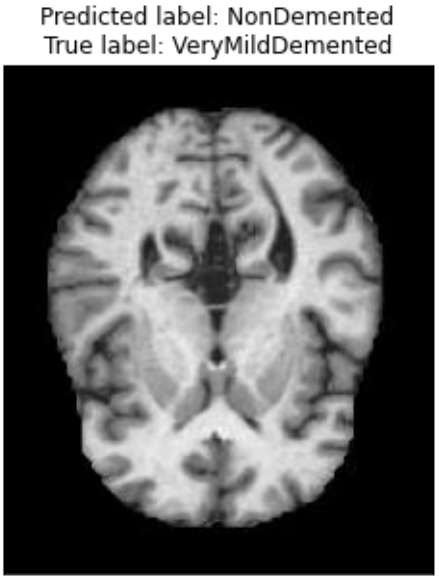


Figure 37: Failed prediction 4 of the first model

As we could already see in the confusion matrix, the failed predictions are similar to the SVM ones. The worst cases are the ones that predict *NonDemented* when the patients do have Alzheimer.

5. Conclusion

Alzheimer's diagnosis is a tedious process that requires several resources. The most important one is time: Alzheimer's may worsen throughout its diagnosis process. It is possible to slow down this deterioration by applying a treatment, which would improve the patient's life quality. Therefore, we wanted to try different techniques so as to make Alzheimer's diagnosis quicker and easier.

As we have already seen, artificial intelligence and precisely machine learning (traditional machine learning and deep learning) have revolutionized the computer vision sector. We can now solve problems that were inconceivable in the past. The purpose of this project is to prove whether machine learning is able to detect Alzheimer's. To achieve this, we have used different techniques (logistic regressions, SVMs and CNNs), which have led to satisfactory results.

First of all, we tried the logistic regression, the SVM, the multinomial Naive Bayes with the default parameters and three CNNs. The multinomial Naive Bayes' accuracy was not high enough, so we discarded it. Initially, the CNNs had higher accuracy. However, the performance of the logistic regression and the SVM got better after the optimization stage, to a point where the SVM surpassed the CNNs.

Alzheimer's is the most common cause of dementia among old people. If its diagnosis was as easy as getting an MRI image and evaluating it, we could improve a high percentage of the patients' lives before it is too late. We have been able to achieve the project's objective following this assumption. Now, we are capable of detecting Alzheimer's with just an MRI image with a very high reliability.

This project would be a useful tool for doctors in order to make the diagnosis process much more efficient.

6. Bibliography

[1] “What Is Alzheimer's Disease?,” NIH National Institute on Aging, 16 May 2017. [Online]. Available: https://www.nia.nih.gov/health/what-alzheimers-disease. [Accessed 7 September 2020].

[2] “How Is Alzheimer's Disease Diagnosed?,” NIH National Institute on Aging, 22 May 2017. [Online]. Available: https://www.nia.nih.gov/health/how-alzheimers-disease-diagnosed. [Accessed 8 September 2020].

[3] J. Brownlee, “What is Deep Learning?,” Machine Learning Mastery Pty. Ltd., 14 August 2020. [Online]. Available: https://machinelearningmastery.com/what-is-deep-learning/. [Accessed 8 September 2020].

[4] “Artificial neural network,” Wikimedia Foundation, Inc., [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neural_network. [Accessed 8 September 2020].

[5] “Anaconda (Python distribution),” Wikimedia Foundation, Inc., [Online]. Available: https://en.wikipedia.org/wiki/Anaconda_(Python_distribution). [Accessed 10 September 2020].

[6] “Jupyter,” Project Jupyter, [Online]. Available: https://jupyter.org/. [Accessed 10 September 2020].

[7] P. Doorwar, “OS Module in Python with Examples,” GeeksforGeeks, 29 November 2019. [Online]. Available: https://www.geeksforgeeks.org/os-module-python-examples/. [Accessed 13 September 2020].

[8] “OpenCV,” Intel Corporation, [Online]. Available: https://opencv.org/about/. [Accessed 13 September 2020].

[9] “NumPy,” Wikimedia Foundation, Inc., [Online]. Available: https://es.wikipedia.org/wiki/NumPy. [Accessed 14 September 2020].

[10] “math — Mathematical functions,” The Python Software Foundation, [Online]. Available: https://docs.python.org/3/library/math.html. [Accessed 6 November 2020].

[11] “csv — CSV File Reading and Writing,” The Python Software Foundation, [Online]. Available: https://docs.python.org/3/library/csv.html. [Accessed 30 September 2020].

[12] J. Hunter *et al.*, “Matplotlib: Visualization with Python,” [Online]. Available: https://matplotlib.org/. [Accessed 6 December 2020].

[13] “itertools — Functions creating iterators for efficient looping,” The Python Software Foundation, [Online]. Available: https://docs.python.org/3/library/itertools.html. [Accessed 28 October 2020].

[14] D. Cournapeau *et al.*, “Scikit-learn,” [Online]. Available: https://scikit-learn.org/stable/. [Accessed 26 September 2020].

[15] S. van der Walt *et al.*, “Scikit-image,” [Online]. Available: https://scikit-image.org/. [Accessed 15 September 2020].

[16] Google Brain Team, “TensorFlow,” Google, [Online]. Available: https://www.tensorflow.org/. [Accessed 3 October 2020].

[17] “Keras,” [Online]. Available: https://keras.io/. [Accessed 3 October 2020].

[18] “Machine Learning Project Structure: Stages, Roles, and Tools,” AltexSoft, 22 February 2018. [Online]. Available: https://www.altexsoft.com/blog/datascience/machine-learning-project-structure-stages-roles-and-tools/. [Accessed 11 September 2020].

[19] Dr. Reisberg, “What Are the 7 Stages of Alzheimer’s Disease?,” A Place for Mom, Inc., [Online]. Available: https://www.alzheimers.net/stages-of-alzheimers-disease/. [Accessed 12 September 2020].

[20] S. Tangaro *et al.*, “Feature Selection Based on Machine Learning in MRIs for Hippocampal Segmentation,” Hindawi, 18 May 2015. [Online]. Available: https://www.hindawi.com/journals/cmmm/2015/814104/. [Accessed 13 October 2020].

[21] S. Dubey, “Alzheimer's Dataset (4 class of Images),” Kaggle Inc., 26 December 2019. [Online]. Available: https://www.kaggle.com/tourist55/alzheimers-dataset-4-class-of-images. [Accessed 12 September 2020].

[22] H. Mishra, “What is a feature descriptor in image processing (algorithm or description)?,” Stack Exchange Inc., 17 July 2018. [Online]. Available: https://stackoverflow.com/questions/27595455/what-is-a-feature-descriptor-in-image-processing-algorithm-or-description. [Accessed 16 September 2020].

[23] “Histogram of oriented gradients,” Wikimedia Foundation, Inc., [Online]. Available: https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients. [Accessed 17 September 2020].

[24] “Module: feature,” [Online]. Available: https://scikit-image.org/docs/dev/api/skimage.feature.html?highlight=lbp#skimage.feature.hog. [Accessed 17 September 2020].

[25] A. Singh, “Feature Engineering for Images: A Valuable Introduction to the HOG Feature Descriptor,” Analytics Vidhya, 4 September 2019. [Online]. Available: https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/. [Accessed 19 September 2020].

[26] “sklearn.model_selection.train_test_split,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. [Accessed 20 September 2020].

[27] M. R. M. Talabis *et al.*, “Supervised Learning,” Elsevier B.V., 2015. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/supervised-learning. [Accessed 19 October 2020].

[28] A. Pant, “Introduction to Logistic Regression,” Towards Data Science Inc., 22 January 2019. [Online]. Available: https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148. [Accessed 20 October 2020].

[29] “Bayes' theorem,” Wikimedia Foundation, Inc., [Online]. Available: https://en.wikipedia.org/wiki/Bayes%27_theorem. [Accessed 22 October 2020].

[30] C. Maklin, “Support Vector Machine Python Example,” Towards Data Science Inc., 12 August 2019. [Online]. Available: https://towardsdatascience.com/support-vector-machine-python-example-d67d9b63f1c8. [Accessed 25 October 2020].

[31] J. Brownlee, “How Do Convolutional Layers Work in Deep Learning Neural Networks?,” Machine Learning Mastery Pty. Ltd., 17 April 2020. [Online]. Available: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/. [Accessed 27 October 2020].

[32] J. Brownlee, “A Gentle Introduction to Pooling Layers for Convolutional Neural Networks,” Machine Learning Mastery Pty. Ltd., 5 July 2019. [Online]. Available: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/. [Accessed 28 October 2020].

[33] Arunava, “Convolutional Neural Network,” Towards Data Science Inc., 25 December 2018. [Online]. Available: https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05. [Accessed 30 October 2020].

[34] “Softmax function,” Wikimedia Foundation, Inc., [Online]. Available: https://en.wikipedia.org/wiki/Softmax_function. [Accessed 30 October 2020].

[35] “sklearn.model_selection.GridSearchCV,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. [Accessed 3 November 2020].

[36] “sklearn.linear_model.LogisticRegression,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html. [Accessed 2 December 2020].

[37] “sklearn.svm.SVC,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html. [Accessed 4 December 2020].

[38] G. Fisher, “Plot a Confusion Matrix,” Kaggle Inc., 12 April 2017. [Online]. Available: https://www.kaggle.com/grfiv4/plot-a-confusion-matrix. [Accessed 7 December 2020].

[39] J. Brownlee, “How to Visualize Filters and Feature Maps in Convolutional Neural Networks,” Machine Learning Mastery Pty. Ltd., 5 July 2019. [Online]. Available: https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/. [Accessed 10 December 2020].